2002

Algorithms to efficiently partition Poisson distributed data

David Foster Barnes
San Jose State University

ALGORITHMS TO EFFICIENTLY PARTITION POISSON DISTRIBUTED DATA

A Thesis Presented to

The Faculty of the Department of Mathematics

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

David Foster Barnes

December 2002

APPROVED FOR THE DEPARTMENT OF MATHEMATICS

Dr. Brad Jackson

Dr. John Mitchem

Dr. Wasin So

APPROVED FOR THE UNIVERSITY

ABSTRACT

ALGORITHMS TO EFFICIENTLY PARTITION POISSON DISTRIBUTED DATA

By David F. Barnes

The objective of this thesis is to evaluate algorithms for finding optimal or near-optimal partitions of Poisson distributed data with respect to an objective function derived using Bayesian statistics. Four basic types will be explored: local search, tabu search, simulated annealing, and dynamic programming. The background information relevant to the problem will be introduced, and each algorithm will be explained in detail. Results of performance comparisons for one and two-dimensional data will be presented. In dimension one, an algorithm that provably finds an optimal solution to the problem using the principles of dynamic programming will be the standard by which all other algorithms are compared. The analysis of algorithms for the two-dimensional problem will focus on improvements over local search results offered by tabu search and simulated annealing.

Acknowledgments

To my wife, Jennifer, for all her support

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction of the Problem

1.1 The Generalized Problem

The goal of this thesis is to evaluate methods that efficiently find the spatial structure a finite set of points, $G = \{p_1, p_2, \ldots, p_n\}$, has in some bounded region B of R^1 or R^2. In other words, it is hoped that the following questions can be answered. Are there clusters, or connected regions, that have a similar distribution of points throughout? If such regions exist what are their boundaries? The hope is not only to answer these questions, but to do so in a reasonable amount of time, and with a minimal amount of computation.

The data itself, and the source of the data, may pose other, more specific questions that can be answered if the spatial structure is known. For example, given a list of non-decreasing real numbers, find segments that contain more numbers from the list than nearby segments. If the list gives the times at which some signal or event was detected the process is similar to discerning a true signal from background noise.

Here, a single objective function is given that will be employed in dimensions one and two in an attempt to find the best partition of a finite set of points into clusters. Each distinct partition will have a corresponding value given by the objective function. The performance of this function with respect to other methods will not be addressed. It is assumed that reasonable results are given in regards to the desired information, so the problem at hand is that of simply partitioning the space to find the largest value possible.

To further simplify the problem, a limitation will be imposed on the manner in which B may be divided up. Partitions must have boundaries that coincide with some subset of the edges given by the Voronoi tessellation of the space relative to the data. Also, these boundaries must generate closed, connected regions. Constructing the Voronoi tessellation of a space based on a finite set of points G amounts to enclosing each point p in a line segment or polygon that also contains every point in the space closer to p than any other point in G. Then only partitions constructed from connected blocks that are unions of Voronoi polygons are considered.

With these limitations the problem is better defined, and reduces to the following exercise in optimization. Find the partition of the space B such that the objective function gives an optimal value based on the number of points of G in, and the volumes of, each disjoint region. This simplifies implementation of algorithms to solve the problems as well, since once the area of each Voronoi interval or polygon is known, the actual data can be discarded, though it is usually not necessary to do this. At times it might be convenient, however, since while searching for a solution computations involving the actual data can be avoided. This method is developed in [Sc01], [Sc99], and [Sc98].

1.2 The Role of Combinatorial Optimization

A few terms and ideas from combinatorial optimization will be outlined to clarify methods used to solve this problem. The *solution space* for this problem is all possible partitions of the data into connected regions that use the edges of the Voronoi tessellation

as the boundaries for each region. Also, a fixed boundary that encloses the data will be imposed so that the space is finite. The region in which the data occurs is bounded, and there exists only one Voronoi tessellation for a fixed set of points, so there are only a finite number of partitions that need to be considered. Call this finite set the set of all *feasible partitions*, and denote it S. As the number of data points grows, the number of feasible partitions grows exponentially. The objective function will be the log of a global posterior function defined in [Sc01]. Denote the objective function by P. With this, the problem can be restated as the ordered pair (S, P), where P is a mapping from the set of all feasible partitions to the real numbers, $P : S \rightarrow R$ [AK].

Ideally, the partition in S that yields the greatest value for P should be found. Call any partition in S by the name Δ, and a partition that yields the optimal value for P by the name Δ_{opt}. In very large or complex problems, only an exhaustive search would be guaranteed to find the optimal partition. In this case, approximations, or solutions that might be close to the best solution should be considered. The exact optimal solution might never be known, but it is hoped the approximate solution is close to it. Denoting the approximate solution by Δ_{approx}, then $| P(\Delta_{approx}) - P(\Delta_{opt}) | \leq \varepsilon$, for some relatively small value ε.

As an algorithm searches the solution space the number of partitions available to be considered as an improvement on the current solution is often limited for the sake of efficiency. An increase in the number of partitions considered in a step corresponds to an increase in the time the algorithm requires to complete that iteration. For any algorithm, a *neighborhood structure* is a mapping

$$N : S \rightarrow 2^S$$

that assigns to each partition Δ a subset S_Δ of partitions that are one iteration of the algorithm from the current partition [AK]. To formalize what solutions are available at each step in an algorithm, the neighborhood structure will be defined as the algorithm is introduced.

The neighborhood structure for a particular algorithm weighs heavily in determining how efficiently, and completely that algorithm can solve, or approximate a solution to, a problem. The method by which an algorithm progresses to the next partition from those in the neighborhood of the current solution has a similar impact on the efficiency and accuracy of an algorithm. The way a partition is chosen from the neighborhood of the current partition will be called the *generation mechanism* for the algorithm [AK].

1.3 Prospective Methods

To find optimal, or near optimal solutions to the problem, a variety of algorithms will be investigated. The fastest results are usually achieved using greedy local search methods, though they typically do not return the best solutions. These concentrate on small parts of the problem, building regions, or breaking them down, in short steps. If at any stage in such an algorithm there is no partition in the neighborhood of the current partition that improves the objective function value, the process terminates. This is analogous to reaching and stopping at a local maximum for a continuous function when searching for its absolute maximum. A partition that yields a better value for the objective function than any partition in its neighborhood is a *local maximum* for a

particular algorithm. One that yields a better value than any other partition in S is called a *global maximum.*

Two techniques that offer a limited solution to the problem of getting trapped at local maxima are simulated annealing and tabu search. These algorithms use the same neighborhood structure as some of the local search methods. Neither guarantees an improvement on the greedy local searches, but both allow progression past a partition that is a local maximum.

In one dimension the problem is structured so that a fairly efficient algorithm that finds the optimal solution exists. This approach uses the principle of optimality in dynamic programming to evaluate and compare a small portion of all possible partitions. This principle states that any subpartition of an optimal partition is optimal [HS]. Each evaluation in a series of steps eliminates the need to compare a number of other solutions. A comparison between the performance of this approach and the other algorithms will be made, but because of its efficiency, there are rarely cases where one of the other algorithms might be a better option. This technique does not adapt well to two, or higher dimensions.

Because the problem in two dimensions does not hold quite as nice a structure as in one dimension, using dynamic programming lessens the amount of work to be done only slightly. As a result, local search algorithms, simulated annealing, and tabu search will all be considered to find near-optimal solutions. A performance comparison similar to that done in one dimension will be done in two dimensions to evaluate these algorithms.

Chapter 2

Voronoi Tessellations and Bayesian Priors

2.1 Voronoi Tessellations

In general, a *tessellation* of a region in R^d is a division of the space into polyhedra [SKM]. Since events in one and two dimensions are considered here, only tessellations of a line into segments and of a plane into polygons need be considered. A more exact description of a tessellation of these is as follows.

Let Π be the set of all bounded non-degenerate segments in R^1 or convex polygons in R^2. A finite subset $\Psi \subset \Pi$ is a tessellation of a bounded region B if:

1. $\psi_i \cap \psi_j = \varnothing$ for all $\psi_i, \psi_j \in \Psi$, $\psi_i \neq \psi_j$
2. $\bigcup_{\psi_i \in \Psi} \psi_i^{cl} = B \subset R^n$, $n = 1$ or 2 where ψ_i^{cl} denotes the segment or polygon ψ_i with its boundary [SKM].

The segments or polygons ψ_i are the *cells* of the tessellation. The endpoints of the segments ψ_i, and the vertices of the polygons p_i are called *nodes* [SKM]. In dimension two, an edge of a cell is a line segment in the set of all boundaries that joins two nodes, and no other nodes lie on that edge [SKM]. In one dimension, the boundaries of the cells coincide with the nodes of the tessellation [SKM].

Given a finite set of points $P = \{p_1, p_2, ..., p_n\}$ in one or two dimensions, nearly all points, $x \in R^i, i = 1,2$, have a unique nearest point $n(x)$ in P [SKM]. The *neighborhood* $N(p_i)$ of a point in P is defined by

$$N(p_i) = \{x \in B : n(x) = p_i\} \text{ [SKM].}$$

These $N(p_i)$ are all open connected intervals, in R^1, or open convex polygons, in R^2. The collection of $N(p_i)$ constitutes what is called the Voronoi tessellation $V(P)$ relative to P [SKM].

For the purposes here, the Voronoi tessellation will be defined as above, adding the boundary of B, a connected, bounded region containing all the points of P, to the edge set that makes up the polygons or intervals of $V(P)$. If the data given is one-dimensional, then B is a closed interval containing all the points of P. If the data is two-dimensional, then B is some closed, connected region in R^2 that, again, contains all the points of P. Thus the neighborhood of a point in P is the set of all points in B closer to p_i than any other p_j in P.

There is a one-to-one correspondence between the actual data points and the Voronoi cells. The goal of this process is to combine cells to create connected regions with nearly uniform density throughout. These larger regions created by combining cells will be called *blocks* [Sc01]. For all methods surveyed, only cells that are natural neighbors, or in other words, share a boundary, may be combined.

These Voronoi diagrams give a reasonable approximation to the relative density of data near a point. Let v_i be the length or area of the cell corresponding to point p_i.

Then $1/v_i$ gives a value that corresponds to the density near p_i [Sc01]. If the length of segment or area of the cell containing p_i is small then it is apparent that there are other points near p_i.

In real data, it is likely that some structure exists; or rather there are some adjacent cells that have a similar value for $1/v_i$. These should be grouped together, or if no structure exists, there needs to be some quantifiable justification to illustrate this. Thus the development of some objective function to quantify how good a partition of cells into blocks relative to other partitions is motivated.

A note on the boundaries used in the implementation of these algorithms. In one dimension, only the first and last points of the data set do not have explicitly defined boundaries. To remedy this problem, the unknown boundaries for these points are taken to be the same distance from the points as the respective known boundaries but in the opposite direction.

In two dimensions the problem of unbounded Voronoi cells is encountered fairly frequently. Another way of describing these unbounded regions, and the manner it is defined in the environment in which the algorithms were implemented, is that such a Voronoi cell has a node at infinity. To fix this problem, a rectangular region, B, containing all of the data is imposed on the space. For each data point, its cell is redefined to be the region contained in the intersection of its Voronoi cell, and the bounding region. This also helps in fixing the problem of cells that lie on the boundary that have extremely large areas. It is also a better estimator of real applications. In many cases, two-dimensional problems arise in searching images or data collected from an

instrument with a limited field of input. Thus imposing this bounding rectangle is reasonable.

2.2 The Development of the Objective Function

Before presenting the objective function, a few preliminaries relating to Bayesian statistics will be outlined. These will give a feel for how the function is derived, and what kinds of assumptions are made on the data. The basis for this field of statistics is Bayes' Theorem, which states: If $\{x_i\}_{i=1}^{n}$ is a set of mutually exclusive events $(\Rightarrow x_i \cap x_j = \varnothing \; \forall \, i \neq j)$, such that $p(x_i) \geq 0 \;\; \forall i$ and the sample space S is equal to the union of all these events, then for any event y in S with positive probability:

$$p(x_j \mid y) = \frac{p(y \mid x_j)p(x_j)}{\sum_{i=1}^{n} p(y \mid x_i)p(x_i)}.$$

This equals $\frac{p(x)p(y \mid x)}{\int_{x} p(x)p(y \mid x)dx}$ for the continuous case . Here, y is the data obtained, or observations made, and x is the previous assumption (or model). The statement

$$p(x_j \mid y) \propto p(y \mid x_j)p(x_j)$$

can also be made [Le].

So that $p(y \mid x\ )$ may be thought of as a function of x, the function

$$l(x\ \mid y) = p(y \mid x\ )$$

is defined to be the *likelihood* function [Le].

Using the form of Bayes' theorem written with a proportionality sign, the likelihood function need only be interpreted as a constant multiple of $p(y \mid x)$. With this, it is sometimes convenient to use the form $\frac{l(A_j \mid B)}{\int l(A_j \mid B) dA_j}$, which is referred to as the *standardized likelihood* [Le]. Replacing the integral in the denominator by a summation accommodates the discrete case.

Bayesian statistics allows for the assignment of a probability to an event prior to its occurrence. When enough data is gathered, the posterior distribution, or the value found after data has been collected, approaches the standardized likelihood. The concept of enough data varies greatly between different applications, so it is heavily dependent upon the context in which the ideas are being applied. The posterior only approximates the standardized likelihood if the prior variance is not infinite, however.

To remedy this problem, an *improper prior* may be used so that the standardized likelihood may be achieved in the posterior [Le]. Taking an improper prior usually amounts to using one that has infinite variance. It is referred to as improper since such an assumption cannot be represented by any true probability density function. A true and proper probability density function will integrate to unity over the whole real line. Assuming that the variance is infinite implies that the data is distributed equally over the entire sample space. This often does not allow for such an integration to occur since the sample space may be the entire real line.

Regardless of this contradiction it is often convenient to use such densities in prior assumptions. In certain cases, taking an improper prior and combining it with an

ordinary likelihood will yield a posterior that is proper [Le]. Doing this results in the likelihood being the dominant feature in the posterior [Le].

It is also reasonable to use a prior that is to be considered valid over a range of values, but invalid outside that range. Lee states that a *locally uniform prior* is one that changes little within the region for which the likelihood is most appreciable, and does not take on large values outside that region. For such a prior $p(x \mid y) \propto p(y \mid x) = l(x \mid y)$, or the normalized posterior is equal to the likelihood [Le]. Taking a uniform prior allows the features of the posterior to be determined almost entirely by the data.

Only the assumption that every physically realizable event is equally likely is made here. All physically unrealizable events are given a prior probability of zero. This is an improper prior but it is one that seems practical. As a consequence, the information given by the posterior will be completely dependent upon the data.

The posterior for a model m_n that attributes a constant Poisson rate λ_n to the data in the region n of the data space is:

$$p(m_n) = f(N,V) = \frac{\Gamma(N+1)\Gamma(V-N+1)}{\Gamma(V+2)} \text{ [Sc01]}. \qquad (1)$$

The equation depends only on N and V, the number of data points in the space and the volume of the space [Sc01]. Scargle makes a note that λ_n does not appear, since a flat, improper prior was taken, making the assumption on the rate of little importance.

For a data set of population n that lies in a region of volume v, the function value given by a model m of the data, where the space is divided into regions $r_1, r_2, \ldots r_k$, with

corresponding volumes and populations $v_1, v_2, \ldots v_k$, $p_1, p_2, \ldots p_k$ respectively, such that $\sum_{i=1}^{k} v_i = v$ and $\sum_{i=1}^{k} p_i = n$, is given by

$$P(m) = \prod_{i=1}^{k} f(p_i, v_i) \text{ [Sc01]}. \tag{2}$$

The product is due to the regions being independent. One of the potential models is of course one that assumes the data has a constant rate, or simply the single model where k=1. Two models with the same number of regions need not be equal. For example, there are many partitions of the data into just two regions. So given a region containing data, the question does the data have a constant Poisson rate, or do sub-regions with distinct Poisson rates exist, can be answered by maximizing the above posterior.

To ease the computations for the methods used to solve the problem, the log of this objective function will be taken. As $f(p_i, v_i)$ involves the gamma function, fairly small values for s_i and v_i will yield very large values for f. Considering several regions of moderate population and volume would yield even greater values in $P(m)$. Taking the natural log of the objective function remedies both of these problems. The log of $f(p_i, v_i)$ returns manageable values for even rather large values for p_i and v_i. Also, $P(m)$ becomes additive over the disjoint regions of a partition rather than multiplicative. So equations (1) and (2) become

$$\ln(p(m_n)) = \ln(f(N, V)) = \ln(\Gamma(N+1)) + \ln(\Gamma(V-N+1)) - \ln(\Gamma(V+2)) \tag{3}$$

and

$$\ln(P(m)) = \sum_{i=1}^{k} \ln(f(p_i, v_i)). \tag{4}$$

For simplicity in notation, and since only this form will be used for the remainder of this paper, $f(p_i, v_i)$ will denote the log of (1), or equation (3), and $P(m)$ will denote the log of (2), or equation (4).

2.3 Defining the Problem

With the objective function explained more explicitly, and the specific class of partition to be used apparent, the problem at hand may be defined more clearly. The spatial structure of the data is to be found using the given tessellation of the space and objective function. Though this approach may seem limiting, all the ideas used are quite natural and make no any heavy assumptions on the data.

Given the data and a bounded region in which it occurs, generate the Voronoi tessellation as described in (2.1). Associate with each Voronoi cell its length, in one dimension, or area, in two dimensions. Call each of these Voronoi cells with its associated population of one and length or area a *cell*. Any partition of cells to create larger blocks that constitute connected regions is a potential model for the data. One hopes to find m_{opt}, the model that yields the greatest value for the objective function, (4).

Another representation of the problem is given by the dual graph of the initial partition. Represent each Voronoi cell with a vertex, and cells that share a boundary are connected by an edge. Call this graph D. Also, associate with each vertex the volume of its corresponding Voronoi cell and a population of one. Rather than creating connected

regions, consider contractions, or combining of adjacent vertices to be represented by a single vertex. In contracting the graph, the new vertex will have a population equal to the number of vertices represented by the new vertex, and a volume equal to the sum of the volumes of all those vertices. Any model would have a corresponding contraction, and hence m_{opt} would have a contraction, D_{opt}. The dual graph and the actual Voronoi diagram will be used synonymously, since both representations are helpful in explaining methods to solve the problem.

Using the graphical interpretation, the problem reduces to contracting the original graph D to represent a feasible partition of the data. In a graph with n vertices, the number of possible contractions grows exponentially as n increases. So though the graphical representation gives an easy representation of the problem, it does not propose an efficient way to search all possible solutions.

Regardless of how the problem is presented, efficient search methods must be explored to find a solution. Local search methods are efficient, but not always accurate. Since they act only on information derived from small portions of the problem there is no guarantee of finding an optimal solution. Also, information on how close the solution might be to an optimal partition cannot be derived. Simulated annealing and tabu search generalize local searches to generate more flexible algorithms. These too fail to guarantee success. In one dimension, an optimal solution can be found fairly efficiently, but the method used does not lend itself to work efficiently in two dimensions. Hence, local search, simulated annealing, and tabu search methods are the most reasonable algorithms to use in finding solutions efficiently in two, and higher, dimensions.

Chapter 3

Local Search Algorithms

3.1 Local Search Methods

Often, considering all possible solutions to a combinatorial optimization problem is not feasible. For the problem addressed here, if n data points are given in one dimension, considering all possible solutions amounts to comparing all 2^{n-1} possible partitions. In this instance of the problem, every data point has at most two neighboring points. As higher dimensions are considered, every point may be adjacent to several points. Consequently, the number of possible partitions also grows, thus increasing the likelihood that an exhaustive search of all possible solutions is impractical. Hence, some other approaches to finding a solution should be explored.

Given the data set P and the Voronoi tessellation of the space based on P introduced in (2.1), call combining two adjacent blocks to create a new block, or dividing the cells in a single block to create two new blocks a *local action*. A *local search algorithm* is a method that attempts to approximate the optimal partition of the data using only local actions that immediately benefit the objective function. Given the proper sequence of progression any partition of the data can be obtained using only local actions. Regardless, it is highly likely that some partitions will not be generated.

The neighborhood of a partition of the data is defined by the generation mechanism of the chosen local search algorithm. A larger neighborhood tends to translate to a greater expenditure of time and resources. In some cases the consideration given to a broader

range of possible partitions proves helpful in obtaining better solutions. In other instances however, this is inconsequential.

Implicit in any local search algorithm is the problem of scope. All possible local actions are limited to merging adjacent blocks, or dividing a single block into two. More specifically, only blocks that share a face may be combined, and a single block can only be split along cells that, again, share a face.

Common Local Search

Step 1 Choose an initial partition Δ_1, of the data

Step 2 Generate acceptance criterions $a(\Delta)$ for all partitions Δ in the neighborhood N_1 of Δ_1

Step 3 $\Delta_t = \Delta_1$

Step 4 While $a(\Delta) > 0$ for some $\Delta \in N_t$ do steps 5-6

Step 5 choose Δ_{t+1} such that $a(\Delta_{t+1}) \geq a(\Delta)$ for all $\Delta \in N_t$

Step 6 compute $a(\Delta)$ for all partitions Δ in the neighborhood of Δ_{t+1}

Step 7 Accept the partition generated by the algorithm and the value it yields for the objective function.

There are several ways one may begin to partition data using a local search algorithm. In some cases, the success or failure of the process may depend greatly on the initial partition chosen. Other times, an algorithm may be equally effective, or ineffective, regardless of this choice. The local search algorithms employed here depend quite heavily on the initial partition of the data.

A *complete* initial partition is one that uses the division of the space into cells as its starting state. Utilizing this starting point might require fewer computations at each stage, since an algorithm could be limited to only combining blocks. With this, a transition might only necessitate accounting for new adjacencies. If dividing blocks is also an option, then this benefit is decreased. The complete partition of the data allows the density of each cell to be assessed and acted on. A single cell of very high or low density may cause an irregularity since it is unlikely that such a cell will be combined with any other cell or block. In one dimension, if this cell lies within a region of rather uniform intensity, the entire region will never be connected to form a single block since local search algorithms are limited to considering only pairs of cells. A *bottom-up method* is an algorithm that starts with a complete partition.

A *single-block* initial partition combines all the elementary blocks to create a single region. Starting in this state all the information regarding the cells from the Voronoi tessellation of the space is retained. From this partition, blocks are divided in an effort to find the optimal partition. In data that favors large clusters, or few groupings, this initial state can present some advantages. When compared to a bottom-up method, which might need to combine many cells before finding a large cluster or group, fewer iterations are required. The problem of scope still exists, but from a different perspective. A single-block start is less likely to encounter problems with small irregularities in the data, but more likely to miss slight changes or accurately model data with many clusters. Algorithms that utilize single-block initial partitions will be referred to as *top-down methods*.

Since at each subsequent stage every possible split of one block to two might be an option, more computations are often required for top-down methods. Even in one dimension, counting every possible division of a block can take a considerable amount of time. Processing data in higher dimensions further heightens the complexity. Some limitation of all possible divisions could be employed to decrease the number of neighbors of a solution. This constraint could itself make the process less effective.

A third class of initial divisions of the data will be called *assumed-state* initial partitions. These are all partitions other than single-block or complete that might be used as a starting state. The initial partition could be determined by assumptions made on the data. For example, an initial grouping seemingly apparent at the outset could be tested or improved upon. Several random divisions might be used in hopes of finding a reasonable approximation of the optimal partition by comparing the results given by the algorithm with these various starts. The limited scope of local search algorithms still proves to be a hindrance even with some knowledge of the structure of the data.

In this setting, a local maximum is a partition such that no local action generates a solution that improves on the objective function. The greatest limitation of local search methods is that if a local maximum is reached, the algorithm will terminate. Since only local actions may be performed, these methods will return a suboptimal solution if some local maximum is reached before finding the global maximum. The limited scope does provide an advantage: at each stage of an algorithm fewer partitions need to be considered, making the algorithm run more quickly.

At every step, it is necessary to have a method to evaluate the objective function value of every partition that is a neighbor of the current state. A more practical approach would be to find the benefit of each possible action on the present partition. So for each action that leads to a neighbor of the current partition, evaluate the change in the objective function. Call this the acceptance criterion for that action. For local searches, only two types need be considered: divide one block into two, or merge two blocks to one.

If the action is to divide a block β_l to create two new blocks, β_j and β_k with volumes v_j and v_k and populations p_j and p_k respectively, the change will be the difference between the sum of the objective function values for the two new blocks and the objective function value for the undivided block. The volume and population for the undivided block may be written as the sum of the volumes and sum of the populations of the two blocks generated by the division, respectively. So the acceptance criterion for dividing a block is:

$$divide\,factor(j,k) = df(j,k) = [f(v_j, p_j) + f(v_k, p_k)] - f(v_j + v_k, p_j + p_k).$$

If the action is to merge two blocks β_j and β_k with the same volumes and populations as above, the change will be the difference between the objective function for the new block, and the sum of the objective functions for the two contributing blocks. Hence,

$$merge\,factor(j,k) = mf(j,k) = f(v_j + v_k, p_j + p_k) - [f(v_j, p_j) + f(v_k, p_k)]$$

is then the acceptance criterion for combining two blocks.

These values give the true effect of the action, which yields the partition Δ on the objective function

$$P(\Delta) = \sum_{I=1}^{k} f(v_I, p_I).$$

If one block is divided in two, the term in the objective function representing that block, $f(v_j + v_k, p_j + p_k)$, will be replaced by two terms, $f(v_j, p_j)$ and $f(v_k, p_k)$, relating to the two new blocks. Similarly, if two blocks are combined, the two terms in the objective function representing those two blocks, $f(v_j, p_j)$ and $f(v_k, p_k)$, will be replaced by a single term, $f(v_j + v_k, p_j + p_k)$, representing the new block. So the additive property of the objective function is useful in eliminating the need to compute its actual value.

In the following sections, several local search methods are explored. Each approach has the limitation that the benefit or detriment of taking an action on only one or two blocks may be considered. In addition to this restriction, limitations specific to each individual method exist. In relation to one another, and to alternative methods, every approach also has advantages. Compared to most search methods that consider taking action on more than one or two blocks, local searches are far more efficient in terms of the amount of information retained and compared at each stage. In addition, local search methods find solutions in a relatively short amount of time.

Methods similar to the first two local search algorithms explored here are mentioned in [Sc01], and outlined in [Sc99].

3.2 The Best Merge Algorithm

The Best Merge algorithm is a local search method that attempts to approximate optimal solutions by only merging blocks. As it is a local search algorithm, only pairs of adjacent blocks may be joined. At each iteration the two blocks that, when merged, have the greatest positive effect on the objective function are found, if any such pair exists. These two blocks are combined, and the process repeats until no combination of blocks will increase the objective function, or only one block exists. Since only merges are performed, this is usually implemented as a bottom-up method.

As the algorithm begins, the volume and population for every block is available. For every pair of adjacent blocks the corresponding merge factor is computed. The largest of these merge factors is found and if it is greater than zero, the two cells are combined. If it is less than zero, then no merging of two blocks will increase the objective function, so the algorithm terminates.

If a merge is performed, the merge factors for all combinations of blocks involving the new block $\beta_{l\cup k}$ are recomputed. Any block β_l adjacent to one of the blocks involved in the merge, β_l or β_k, is now adjacent to $\beta_{l\cup k}$. Merging β_l with β_l or β_k is no longer possible, so it is necessary to evaluate the benefit of merging β_l with $\beta_{l\cup k}$ and consider that as an option in the algorithm.

When all merge factors are less than or equal to zero, the algorithm terminates and returns the current partition of the data.

Best Merge Algorithm

Step 1 Find the volume and population (v_j, p_j), associated with the j^{th} cell

Step 2 Choose an initial partition of the data

Step 3 Generate the merge factors $mf(j,k)$ for each pair of adjacent blocks β_j, β_k.

Step 4 While $mf(j,k)>0$ for some β_j, β_k do steps 5-7

Step 5 Find β_j, β_k such that $mf(j,k) \geq mf(m,n)$ for all m,n

Step 6 Merge β_j, β_k

Step 7 Compute $mf(l, j \cup k)$ for any β_l now adjacent to $\beta_{j \cup k}$

Step 8 Return the partition generated by the algorithm and the value it yields for the objective function

3.3 Best Divide Algorithm

The best divide algorithm is typically implemented as a top-down approach to partitioning data, though any partition may be used as the starting state. Again, as with all local search methods, the data is initially partitioned into cells. This separation is used to make distinctions between portions of blocks. At any stage only a single division can be made: the split of one block into two that will create the greatest positive effect on the objective function is found, and the block that contains this split is divided. When no division of any block yields a positive effect on the objective function, the process terminates.

In dimension one there is exactly one division of a block for every pair of adjacent cells j and $j+1$. As the dimension of the space in which the data occurs increases it might be advantageous to limit the divisions to some feasible subset of all possible divisions to maintain the benefits local search methods provide. In higher dimensions, the number of adjacencies for a single cell is likely to increase greatly, so the number of possible divisions of a single block into two will also increase. The limits imposed on the choices would allow a greater measure of efficiency to be maintained. Once this issue has been addressed the algorithm can be initiated.

The divide factor for every feasible division is computed using the information generated by creating the complete partition of the data. The initial partition is chosen and the largest divide factor is found. If it is greater than zero, the division modeled by that divide factor is performed. If no divide factor exists that will increase the objective function, the algorithm terminates. When implementing these algorithms, the fact that $df(j,k) = -mf(j,k)$ may prove to be useful.

If some division is performed, the divide factors for all feasible subdivisions of each new block β_j, β_k generated by splitting $\beta_{j\cup k}$ must be recomputed. The scope of the algorithm does not allow for such divisions to be evaluated before $\beta_{j\cup k}$ is divided, so the effect of any subdivision of β_j or β_k is available only after the initial division has occurred.

When no division yields any immediate benefit to the objective function the algorithm terminates and returns the current partition of the data.

Best Divide Algorithm

Step 1 Find the volume and population (v_j, p_j), associated with the j^{th} cell

Step 2 Choose an initial partition of the data, and define all feasible divisions

Step 3 Generate the divide factors $df(j,k)$ for every feasible division

Step 4 While $df(j,k)>0$ for some $\beta_{j\cup k}$ do steps 5-7

Step 5 Find $\beta_{j\cup k}$ such that $df(j,k) \geq df(m,n)$ for any other subdivision of $\beta_{j\cup k}$ to β_m and β_n or a division of any other block $\beta_{m\cup n}$

Step 6 Divide $\beta_{j\cup k}$

Step 7 Find $df(l,q)$ for feasible divisions of β_j to β_{j_1} & β_{j_2} and β_k to β_{k_1} & β_{k_2}

Step 8 Return the partition generated by the algorithm and the value it yields for the objective function

3.4 Best Merge/Divide Algorithm

Merge divide algorithms combine the Best Merge and the Best Divide approaches to allow more flexibility in creating blocks of data. Regardless of what partition is used at the start, throughout the algorithm the benefit of merging any two blocks, or making a feasible division of any one block is evaluated. The approach is to perform whatever single action will provide the greatest immediate benefit to the objective function. Again, when no single action will provide an increase for the objective function, the algorithm terminates.

Any initial partition of the data is reasonable since both divisions and merges are possible. For the complete initial partition, though local maxima will still cause the algorithm to terminate, at every stage more actions on the current partition can be considered. Errors that might have been created in the Best Merge algorithm can be corrected in the merge/divide process that has the same starting point. Similarly, in comparison with the divide only algorithm, merge/divide has the ability to correct errors that might be made in the earlier stages when starting with a single block partition. Random blocks of data, or some approximation to a likely combination of cells using heuristics are both reasonable starting points since two actions are available to the algorithm.

The initial computation to evaluate the benefit of all possible actions depends on the partition of the data chosen at the start of the process. Subsequent computations required at each iteration depend on if a block was divided, or if two blocks were merged. The same merge and divide factors, $mf(j,k)$ and $df(j,k)$ as described in the previous sections are employed by this method. For every possible combination of two blocks β_j and β_k, $mf(j,k)$ is computed. Similarly, for any feasible division of a block $\beta_{j\cup k}$ $df(j,k)$ is generated. These will be combined here and in later sections as a merge/divide factor, $m/df(j,k)$

The evaluation of possible actions may depend on the limitations imposed on the types of divisions allowed. Just as in the previous two algorithms, at every stage the action that will provide the greatest benefit is carried out, and any new possible actions that appear

as a result are accounted for. Once no action provides any benefit to the objective function the algorithm terminates.

Best Merge/Divide Algorithm

Step 1 Find the volume and population (v_j, p_j), associated with the j^{th} cell

Step 2 Choose an initial partition of the data, and define all feasible divisions

Step 3 Compute $m/df(j,k)$ for all possible merges and all feasible divisions of the initial partition

Step 4 While $m/df(j,k)>0$ for some β_j, β_k, or $m/df(j,k)>0$ for some $\beta_{j\cup k}$ do steps 5-7

Step 5 Find $m/df(j,k) \geq m/df(m,n)$ as described in step 4 of the previous two algorithms

Step 6 Merge or divide the data as prescribed by the merge/divide factor returned in Step 5

Step 7 Compute any merge or divide factors affected by the action done in Step 5

Step 8 Return the partition generated by the algorithm and the value it yields for the objective function

Chapter 4

Approximation Algorithms

4.1 Approximation Algorithms

The merge/divide version is the most versatile, and hence most useful of the local search algorithms. It is still quite limited though, since if any local maximum is reached the process terminates. The investigation of efficient algorithms capable of escaping local maxima is thus motivated. Two such algorithms, tabu search and simulated annealing, will be evaluated and compared with the results given by local searches. The neighborhood structure for these will be the same as that of the merge/divide local search.

Both of these will be called *approximation algorithms*, since there is no guarantee of reaching an optimal solution, or even improving local search results. They do, however, utilize generation mechanisms that allow progression beyond local maxima in an attempt to give a better approximation of an optimal solution. Tabu search relies on the idea that local maxima are fairly close together. In other words, only a few actions separate two local maxima. Allowing a local search algorithm to check the neighborhoods of a few solutions near a local max could then result in finding a better partition. Simulated annealing uses a well-directed method of randomly choosing a solution from those in the neighborhood of the current state.

4.2 Tabu Search

Tabu search differs from local search methods only in that when a local maximum is reached, the algorithm continues. When no solution in the neighborhood of the current one yields an improvement on the objective function, the one that provides that smallest decrease is picked to be the next partition. That is, the action that corresponds to the largest merge or divide factor, regardless of sign, is performed. Progressing in this manner, it is hoped that at some point a solution is encountered that improves on the one returned by the standard merge/divide local search.

As this algorithm searches the solution space some local maximum is moved beyond. Every partition in the neighborhood of that local maximum has a smaller corresponding function value. As a result, undoing the action that was just performed and returning to the local maximum will certainly be beneficial. To avoid such a situation, a running list of the last t actions performed is retained. These actions are referred to as *tabu*, or invalid, and are retained in a *tabu list*. The length of the tabu list can be varied to attempt improving the output of the algorithm. Any action that would be available in the best merge divide algorithm save those on the tabu list may be performed.

As local maxima will not terminate the algorithm, some other stop criterion is needed. Here, a limit on the number of iterations is imposed. This may be changed to accommodate the size of the data, time constraints, or allow longer runs. Since the algorithm does not necessarily stop at a maximum, the last partition found might not be the best one encountered during the search. To ensure that the best partition found is the one returned a record of solutions is kept throughout the algorithm. Naturally, only those

that improve the objective function are retained. When a partition is found that gives a higher value than all those found previously, it is recorded as the new best solution.

Tabu Search Algorithm

Step 1 Choose the stop criterion C, and the length t of the tabu list

Step 2 Find the volume and population (v_j, p_j), associated with the j^{th} cell

Step 3 Choose an initial partition of the data, and define all feasible divisions

Step 4 Compute $m/df(j,k)$ for all possible merges and all feasible divisions of the initial partition, set $j = 1$.

Step 5 While $j < C$ do steps 5-10

Step 6 Find $m/df(j,k) \geq m/df(m,n)$ for all merge and divide factors

Step 7 Merge or divide the data as prescribed by the merge/divide factor returned in Step 6

Step 8 Compute any merge or divide factors affected by the action done in Step 7

Step 9 Append the tabu list by making the first entry the undoing of the action just performed, and delete the t^{th} entry from that list.

Step 10 If the new partition has a higher objective function value than all previous partitions, record it as the new best partition, $j = j + 1$.

Step 11 Return the partition generated by the algorithm and the value it yields for the objective function

4.3 Simulated Annealing

The Simulated Annealing algorithm is also very similar to the merge/divide local search. For this algorithm, rather than choosing the solution with the largest merge factor and taking that as the next partition, one is picked at random. The action prescribed is not necessarily carried out, but rather is only proposed and is actually performed in accordance with an acceptance probability. The probability of performing non-beneficial actions is controlled by two parameters declared as the algorithm starts, and decreases as it progresses.

The first of these parameters is called the *control parameter*, denoted c_0 [AK]. This declares an initial state for the system, and changes as the algorithm runs. This initial state helps to determine how long detrimental actions will be accepted with a high probability.

The second parameter is the *decrement constant*, α, which is a value usually less than but close to 1 [AK]. It is used to change the control parameter recursively though the decrement function

$$c_{i+1} = \alpha \cdot c_i .$$

The values for α usually lie between .8 and .99, but are not restricted to this range [AK]. When the algorithm terminates the control will be very close to zero.

A series of actions is allowed for each value of the control. Each series is referred to as a *chain*, because of similarities to finite Markov chains. The lengths of these chains may be varied, and the length of the k^{th} chain is denoted L_k. Simulated annealing may

also be implemented with $L_k = 1$, for every k, though typically this does not return good results.

The last parameter is the stop criterion, or simply the limit C on the number of chains. There is some interaction between the first two parameters and the number of iterations necessary for the algorithm to perform well.

The specification of all these parameters can be summarized in a *cooling schedule*, which indicates the initial value for the control parameter, the decrement constant, the length of each of the chains, and the stop criterion. The variety of cooling schedules available allows a great degree of flexibility in adapting this algorithm to different data sets.

Let the algorithm be at a stage where the i^{th} control parameter is being used, and a particular solution Δ is the current state. Choosing a solution at random in the neighborhood of Δ with corresponding merge/divide factor $m/df(h,j)$, the merge or divide is carried out based on the following acceptance probability:

$$P(accept(a_i)) = \begin{cases} 1, m/df(i,j) > 0 \\ e^{\frac{m\ df(h,j)}{c_i}}, m/df(i,j) < 0 \end{cases} \quad \text{[AK]}.$$

So blocks are always merged or divided if it benefits the objective function, and merged or divided with probability $e^{\frac{m\ df(h,j)}{c_i}}$ otherwise. In implementing simulated annealing, when a negative merge/divide factor is returned, a random number r between 0 and 1 is picked. If $0 \le r \le e^{\frac{m\ df(h,j)}{c_i}}$ then the prescribed action is carried out [AK].

Early on in the algorithm, the values for c_i are fairly high, so some merges and divides that decrease the objective function are performed. This allows some partitions to be visited that otherwise would not be reached by the local search algorithms. As the control is decreased, the value for $e^{\frac{m\_df(h,j)}{c_i}}$ also becomes small, making it less likely that actions that decrease the objective function will be carried out. In the early stages, some of the actions carried out with negative merge/divide factors are hoped to later help in moving beyond local maxima. Those that prove to not be so beneficial will likely be undone, since their merge/divide factors will consistently be greater than zero.

To make sure all actions that increase the objective function are carried out, the length of all the chains and the number of chains must be such that every beneficial action available at the end of the algorithm is performed. Actions are chosen at random, so the stop criterion must allow enough operations so these beneficial merges or divides can be found. Some of the benefits will be the reversal of actions done when the acceptance probability was high for negative merge/divide factors.

A balance must be found between the length and number of chains, and the amount of time acceptable for the algorithm to run. Simulated annealing can be shown to converge asymptotically to the set of all optimal solutions for an instance of a combinatorial optimization problem, see [AK]. So given an infinite amount of time, an optimal solution can be found with correct implementation. Though this does not directly help in an applying this technique, it does show that over time, the algorithm comes closer to the optimal solution.

Running the algorithm for some finite amount of time, or rather limiting C to a reasonable number, will give a solution that is fairly close, possibly equal, to an optimal one [AK]. This close approximation depends heavily on the cooling schedule. The decrement on the control must be small enough that some non-beneficial changes are made later in the run so some local maxima may be overcome. It should not be so small that such changes are still being made as the algorithm terminates. If the control is decreased quickly, the chains can be short, and the algorithm can terminate quickly [AK]. Better results are often found when the control is decreased slowly. This requires longer, and often more, chains to make certain only actions increasing the objective function are done near the end of the run.

The initial value for the control parameter also has a heavy impact on the cooling schedule. A high initial control value requires either a large decrement constant or a large number of chains, so again, only beneficial merges or divides are performed as the algorithm nears termination. High initial values for the control can help in allowing more merge/divides to be done early on, which can help to move beyond local maxima.

Simulated Annealing Algorithm

Step 1 Declare a cooling schedule: $[c_0, \alpha, L_k, C]$

Step 2 Find the volume and population (v_j, p_j), associated with the j^{th} cell

Step 3 Choose an initial partition of the data, and define all feasible divisions

Step 4 Compute $m/df(j,k)$ for all possible merges and all feasible

divisions of the initial partition, set $j = 1$.

Step 5 While $j < C$ do steps 6-10

Step 6 While $i < L_j$ do steps 7-9

Step 7 Pick a random merge/divide with merge/divide factor $m/df(h, j)$

Step 8 If $m/df(h, j) > 0$ merge or divide blocks h and j

otherwise merge or divided if $e^{\frac{m\_df(h,j)}{c_i}} > random\ number \in [0,1)$.

Step 9 Compute any merge or divide factors affected by Step 7, $i = i + 1$

Step 10 $c_{i+1} = \alpha \cdot c_i, j = j + 1$.

Step 11 Return the partition generated by the algorithm and the value it yields for the

objective function [AK]

Chapter 5

An Algorithm to Find the Global Optimum

5.1 Dynamic Programming

In one dimension dynamic programming provides an efficient method for finding an optimal partition of the data. Though this algorithm can be extended to two dimensions, it is likely to be very inefficient. Given a set of N data points, this algorithm will have N major stages, with a number of comparisons to be made within each stage. The number of comparisons grows at a fairly slow rate in one dimension. Two-dimensional tessellations tend to yield more natural neighbors for each cell, and as a result many more comparisons are necessary.

Dynamic programming is built on the principal of optimality: an optimal partition of the data is made up of optimal sub-partitions [HS]. Here, sub-partition does not refer to an arbitrary set of connected points, but a single block, or collection of blocks, in the optimal partition.

Lemma 5.1

Suppose the optimal partition Δ_{opt}, of a set of data is known, and S is a sub-partition consisting of blocks $\{B_j\}_{j=1}^{k}$, of Δ_{opt}. Then no other partition of the cells in S yields a higher value of the objective function.

Proof:

Denote the term $f(s_j, v_j)$ in the objective function corresponding to the block B_j. $f(B_j)$. Let R represent the value of the objective function over all the blocks not in S. Then the global maximum for the objective function is given by

$$P(\Delta_{opt}) = R + \sum_{j=1}^{k} f(B_j).$$

Given any other partition of S into blocks $\{B'_j\}_{j=1}^{l}$. the corresponding value for the objective function would be

$$P(\Delta_{new}) = R + \sum_{j=1}^{l} f(B'_j)$$

But.

$$P(\Delta_{opt}) \geq P(\Delta_{new})$$

$$\Rightarrow R + \sum_{j=1}^{k} f(B_j) \geq R + \sum_{j=1}^{l} f(B'_j)$$

$$\Rightarrow \sum_{j=1}^{k} f(B_j) \geq \sum_{j=1}^{l} f(B'_j).$$

Therefore the optimal partition gives the largest objective function value for any partition of the cells of S. □

Using the idea proved in lemma 5.1 the optimal partition of a set of data can be found. The algorithm depends only on the additive behavior of the objective function over independent blocks. So different types of data with varying distributions may be

analyzed with the same technique using other objective functions, so long as the blocks remain independent.

Since the structure of the one-dimensional problem lends itself to being solved fairly efficiently by this method, and the structure of the two-dimensional problem does not, these will be presented in separate sections. The two dimensional algorithm will not be presented as completely, but the factors that lead to its infeasibility will be examined.

5.2 The Algorithm in One Dimension

The one-dimensional algorithm is due to Dr. Brad Jackson and work done by a CAMCOS (Center for Applied Mathematics and Computer Science) team at San Jose State University. For a detailed report of this project, see [J].

The data given is initially partitioned into cells as before. Let the cells be labeled in accordance with the value of the data point to which they correspond. Cell one corresponds to the data point with smallest value, the second cell to the data point with the next largest value, and so on. The following notation will be useful in detailing the algorithm. First, the optimal, or best partition of the first j cells, or cells $1,\ldots,j$, will be denoted $best(j)$. The merge that contains cell i as its last cell, and begins at cell $j+1$ will be denoted $end_i(j+1)$. The subscript i may be dropped since the index of the last cell in the merge will usually be obvious. The objective function value for $best(j)$ will be $P(best(j))$. Also, the objective function value for $end_i(j+1)$ will be $f(end_i(j+1))$.

Finally, the best partition on the first 0 cells $best(0)$, will be useful. The objective function value for this will be defined to be $P(best(0)) = 0$.

The algorithm begins by considering the cell that corresponds to the least, or first, value in the data set. The optimal partition on this one cell is found, trivially, and retained as the best partition for the first one cells, or $best(1)$.

Next, the neighbor of the first cell is added to the process and the optimal partition is found for the first two cells. Two partitions need to be compared. This is due to the existence of only two *end merges*, or merges that include the second cell. First, merging the last cell with no other cell, or the end merge starting at cell 2 $end_2(2)$. Second the end merge starting at cell 1, $end_2(1)$. The values for the objective function correspond to the values for the following two models.

$$P(model(1)) = P(best(1)) + f(end_2(2)) \tag{1}$$

$$P(model(2)) = P(best(0)) + f(end_2(1)) \tag{2}$$

The model that corresponds to the greatest of (1) and (2) will become $best(2)$ and that can be used to find $best(3)$. The optimal partition of the first three cells is given by the following:

$$best(3) = model(i) \text{ such that}$$

$$P(model(i)) = \max\begin{Bmatrix} P(model(1)) = P(best(2)) + f(end_3(3)) \\ P(model(2)) = P(best(1)) + f(end_3(2)) \\ P(model(3)) = P(best(0)) + f(end_3(1)) \end{Bmatrix}. \tag{3}$$

One of $end_3(i)$ $1 \le i \le 3$ will occur in the optimal partition of the first three cells. By lemma 5.1, the remaining cells must be partitioned optimally. Hence only the three

possible models need be compared in (3) since no other partition could possibly return a higher objective function value.

This process can be extended recursively to any number of data points. If partitions $best(i)\ \ 1 \le i \le j$ are all known $best(j+1)$ can be found by comparing the following $j+1$ models:

$$best(j+1) = model(i) \text{ such that}$$

$$P(model(i)) = \max\left\{\begin{array}{l} P(model(1)) = P(best(j)) + f(end_{j+1}(j+1)) \\ P(model(2)) = P(best(j-1)) + f(end_{j+1}(j)) \\ \ldots \\ P(model(j+1)) = P(best(0)) + f(end_{j+1}(1)) \end{array}\right\}. \qquad (4)$$

The order of the algorithm becomes apparent from these examples. Given N points, the optimal partition can be found in N stages, making j comparisons at the j^{th} stage. So the number of comparisons to be done for N data points is $\frac{N(N+1)}{2}$. Though this is not highly efficient, it is far better than searching all 2^{N-1} possible combinations. The illustration of the algorithm assumes N data points are given.

Dynamic Algorithm in One Dimension

Step 1 Find the volume and population (v_j, p_j), associated with the j^{th} cell

Step 2 $P(best(0)) = 0$

Step 3 For $R = 1...N$ do

$best(R) = model(i)$ such that

$$P(model(i)) = \max_{1 \le j \le R} \{P(best(j-1)) + f(end_R(j)\}$$

Step 4 Return the partition $best(N)$ and the value it yields for the objective function

5.3 The Algorithm in Two Dimensions

As was stated earlier, this approach is far less practical in two dimensions. The key to the efficiency in one dimension is that each cell has at most two neighbors. As a cell is added to the process, it is adjacent to exactly one cell that has already been added. This property is lost in two, and higher, dimensions. The graphical interpretation of the problem is convenient to use in describing the algorithm.

The optimal partition of the vertices of a graph corresponding to a two-dimensional Voronoi tessellation, as described in (2.1), can be found in much the same way as in dimension one, but with considerably more computation. For all of the following, the vertices representing the Voronoi cells are labeled so that vertex v_i is adjacent to some vertex v_j such that $1 \le j < i$. Vertex v_i could be part of any contraction that includes one of its previously added neighbors $\{v_{i_1},...v_{i_k}\}$. So the set of end merges, or end contractions that needed to be considered in one dimension become the union of all

contractions that involve only v_i or v_j and one or more of $\{v_{i_1},....v_{i_k}\}$. Call this union $C(v_i) = \{\omega_1,....,\omega_{zz}\}$. At any stage, the optimal contraction of every possible connected subset that constitutes the complement of any of the ω_l in $C(v_i)$ must also be known.

In one dimension, the algorithm actually checks every possible contraction. Since the graph for a one-dimensional data set is simply a path, there are only $\frac{N(N+1)}{2}$ of these. A similar count for the number of possible contractions in a two-dimensional graph might help to illustrate the inefficiency of the algorithm when the data does not present a high degree of structure in its graph.

Suppose a graph D has n vertices that make up its vertex set $V(D)$. The set of all contractions of which a vertex u_i is a member is $C(u_i) = \{\omega_{i1},.....\omega_{ik}\}$. Suppose for some connected subset $V' \subset V(D)$ $C(u_i)$ is know for every $u_i \in V'$. The total number of contractions involving vertices in V' is denoted $N(V')$. Let v be a vertex not in V', but adjacent to one or more vertices $u_1,....u_k$ in V'. As v is added to V' it can generate a new contraction by itself, or from any involving one or more of $u_1,.....u_k$. Thus a recursion relation describing the number of total contractions for a set of connected vertices can be derived. H denotes the number of contractions generated by v and some set of disjoint contractions each containing one or more of $u_1,.....u_k$ not counted otherwise.

$$N(V' \cup v) = |C(v)| + N(V') \tag{5}$$

$$= (1 + N(V') - N(V' \setminus \{u_1,.....u_k\}) + N(V') + H \tag{6}$$

$$= 2N(V') - N(V' \setminus \{u_1,....,u_k\}) + 1 + H \tag{7}$$

In the algorithm in one dimension, vertices are added in a way so that they have only one neighbor in the previous vertex set. This limits the number of contractions a new vertex v can create, since if v and some vertex already in the set are both to be part of some contraction, all those vertices that lie on the path between them must also be part of that contraction. This is not so in two dimensions. While the relation (7) has quadratic growth in one dimension, it grows far faster in two and higher dimensions.

The count given by (7) gives a lower bound for the number of partitions to consider in total. If any algorithm did not check even one possible contraction, a counterexample could be formulated where that contraction was part of the optimal solution. Just as in one dimension, as a vertex is added every possible contraction involving that vertex must be considered. This number is given by

$$N(V') - N(V' \backslash \{u_1, \ldots, u_k\}) + 1 + H \,. \tag{8}$$

Also, the optimal contraction of the remaining vertices must also be retained at every stage. So the amount of information that needs to be available is given by

$$2(N(V') - N(V' \backslash \{u_1, \ldots, u_k\}) + 1 + H) - 1 \,. \tag{9}$$

The subtraction of one in (9) is because the contraction of all vertices to one has an empty complement set.

The set of contractions that involve the vertex v being added to V' is $C(v) = \{\omega_i\}_{i=1}^{k}$. Designate $V(\overline{\omega_i})$ to be the complement set of vertices for the contraction ω_i. Let $best(V(\overline{\omega_i}))$ be the optimal contraction of $V(\overline{\omega_i})$. Thus the optimal contraction of $V' \cup v$ is the model having an objective function value equal to the following:

$$\max_{\omega_i \in C'(v)} \{P(best(V(\overline{\omega_i})) + f(\omega_i)\}. \tag{10}$$

If v is not the last vertex to be added, then it will likely be part of one or more complement sets for a contraction involving a vertex added subsequently. Let V'_1 be the subset of vertices in V' that have neighbors not in V', and $C(v)'_1$ be the set of contractions involving v, and not involving at least one vertex from V'_1. Then the optimal contraction for every element of $C(v)'_1$ must be found if it is not already known.

When a vertex v is added to the process, the optimal contraction of $V' \cup v$ can be found in much the same way as was done in one dimension. If v is not the last vertex to be added, then optimal contractions of several, possibly many, subsets of connected vertices must also be found.

Dynamic Algorithm in Two Dimensions

Step 1 Find the volume and population (vol_j, pop_j), associated with the j^{th} cell

Step 2 Label the vertices in the graph D corresponding to each cell so that v_i is adjacent to some vertex v_j with $1 \le j \le i$

Step 3 $V' = \varnothing$

Step 4 $P(best(V')) = 0$

Step 5 For $i = 1...N$ do steps 6-11

Step 6 $C(v_i) = \{\omega_j : v_i \in \omega_j\}$

Step 7 $best(V' \cup v) = model(i)$ such that

$$P(model(i)) = \max_{\forall \omega_j \in C(v_i)} \{P(best(V'(\overline{\omega_j}))) + f(\omega_j)\}$$

Step 8 $V'_1 = \{u_j$ in $V' : u_j$ has at least one neighbor not in $V'\}$

Step 9 $C(v_i)'_1 = \{\omega_j : v \in \omega_j$, at least one vertex of $V'_1 \notin \omega_j\}$

Step 10 For every $\omega_j \in C(v_i)'_1$, find $best(\omega_j)$ if it is not already known

Step 11 $V' = V' \cup v$

Step 12 Return the partition $best(V)$ and the value it yields for the objective function

Chapter 6

Algorithm Performance Comparison in One Dimension

6.1 Results in One Dimension

Though the optimal solution to the problem is attainable it is worthwhile to compare the performance of various algorithms to give some insight as to how these will perform in two, and higher, dimensions. Also, in the case of a very large data set in one dimension, it may be more practical to use one of the sub-optimal algorithms since the optimal solution is found by an $O(N^2)$ method. The term *change point* will be used in reference to the boundaries of a block. Various algorithms may return different change points for a given set of data, so this return value will help to illustrate which sub-optimal algorithms model data best.

Two types of simulated data will be used to compare the algorithms. Random data will help show on average how close the algorithms approximate the objective function value and the change points given by the dynamic algorithm. This data will also be used to compare how quickly each algorithm is able to find a solution. The second type is data with a known structure. This will be processed to compare how well each algorithm approximates a known model.

Since the optimal solution can be found, variations on some of the algorithms will not be explored in any great depth. These results are intended to illustrate in general how each algorithm should perform. The variety of starting states for the local search merge/divide, simulated annealing, and tabu search algorithms will not be utilized to

improve results. In addition only a very limited number of cooling schedules and tabu list lengths will be shown for the last two of those algorithms.

All the algorithms will be assessed in their performance with respect to the dynamic algorithm. In one dimension, this algorithm will be referred to as *dynamic1d*. The time to terminate for the local search methods will be compared to illustrate the linear increase with respect to the size of the data. The quadratic increase in the dynamic algorithm will be presented as well. Times for the approximation algorithms will be compared to show the effect of changing parameters, rather than changes in the size of the data. Some sacrifice in time is accepted implicitly in each of these since they are by nature less efficient than all the local search methods. The accuracy of all methods will be measured by two parameters. First, how closely the algorithms approximate the objective function value given by dynamic1d. Second, how many of the change points found by the optimal algorithm are found by each of the other algorithms. With this, the number of false and missed change points can also be found and their meaning assessed.

6.2 Local Search Algorithms

Four local search algorithms will be surveyed. *BM1* and *BD1* are implementations of the best merge algorithm and the best divide local search algorithms, respectively. *BMD1c* and *BMD1s* are best merge/divide local search algorithms, starting with a complete partition, and a single block partition respectively. To help in describing the test data, the term overall average rate, or average rate will be used to describe the frequency of occurrence of data in the interval that contains the entire data set. For

example, if a test vector consists of completely random data with an average rate of λ, this means the size of the data set divided by the total length of the interval in which they occur is equal to λ.

The following comparisons were made by generating thirty one-dimensional data sets of the same size all with an overall average rate of approximately .05, with some slight variation between individual vectors. A single rate was chosen so there existed fewer differences between the data being processed, thus yielding a better venue in which to compare the performance of the algorithms. This process was performed for eleven different sizes of data: 100, and 1000 to 10,000 in increments of 1000.

Table 6.1 shows the average times for each algorithm to process data of the given size. The data sets of size 100 were evaluated to show that for small sets the dynamic algorithm finds a solution more quickly than any of the local search methods. As the sizes of the data sets increase above 2000 the time required for dynamic1d to find a solution is greater than any of the others.

The data sets of sizes 1000 to 10,000 were generated to show the increased time required for each algorithm to find a solution as the size of the sets becomes larger. These results are plotted in figure 6.1. The linear increase for local search methods, and the quadratic increase for dynamic1d are both suggested by this plot. For data of the given sizes, these properties are further verified by Table 6.2, which shows the increase in the average times for the various algorithms as the size of the data grows. The local search algorithms show very slow, or little, growth in the difference between the times. This implies the rate of change is near constant, or the increase in actual time is linear. In

contrast, dynamic1d shows moderate change in the average time required to find a solution. The values presented in the table suggest quadratic growth as they are near to the sequence of odd integers given by the difference between consecutive integer values for the equation $f(x) = x^2$.

Table 6.1 Average Times (in seconds) Over 30 Vectors of the Same Size for Algorithms to Find Solutions

Data Size	Algorithm				
	dynamic1d	BM1	BD1	BMD1c	BMD1s
100	0.03	0.04	0.04	0.16	0.04
1000	1.26	0.56	0.50	1.75	0.51
2000	4.61	1.39	1.22	3.61	1.22
3000	10.23	2.50	2.40	5.57	2.42
4000	18.15	3.91	3.41	7.77	3.44
5000	29.31	5.78	5.09	10.20	5.14
6000	41.45	7.77	7.94	12.67	8.00
7000	56.79	10.12	9.42	15.31	9.53
8000	74.48	12.98	12.17	17.87	12.36
9000	94.48	16.40	18.37	20.85	18.48
10000	122.64	20.74	23.75	24.25	23.92

Table 6.2 Differences for Consecutive Times in Table 6.1 Sizes 1000 to 10,000

Difference between times	Algorithm				
	dynamic1d	BM1	BD1	BMD1c	BMD1s
T(2000)-T(1000)	3.35	0.83	0.72	1.85	0.70
T(3000)-T(2000)	5.62	1.11	1.18	1.97	1.20
T(4000)-T(3000)	7.92	1.41	1.02	2.20	1.02
T(5000)-T(4000)	11.16	1.86	1.67	2.43	1.70
T(6000)-T(5000)	12.14	1.99	2.85	2.47	2.87
T(7000)-T(6000)	15.34	2.35	1.49	2.65	1.53
T(8000)-T(7000)	17.69	2.86	2.74	2.56	2.83
T(9000)-T(8000)	20.00	3.41	6.20	2.98	6.12
T(10000)-T(9000)	28.15	4.34	5.38	3.40	5.44
Average Increase	13.49	2.24	2.58	2.50	2.60
Total Increase	121.38	20.18	23.25	22.50	23.41

(T(2000)-T(1000) refers to the difference between the average time for the algorithm to process data of size 1000 and the time to process data of size 2000)

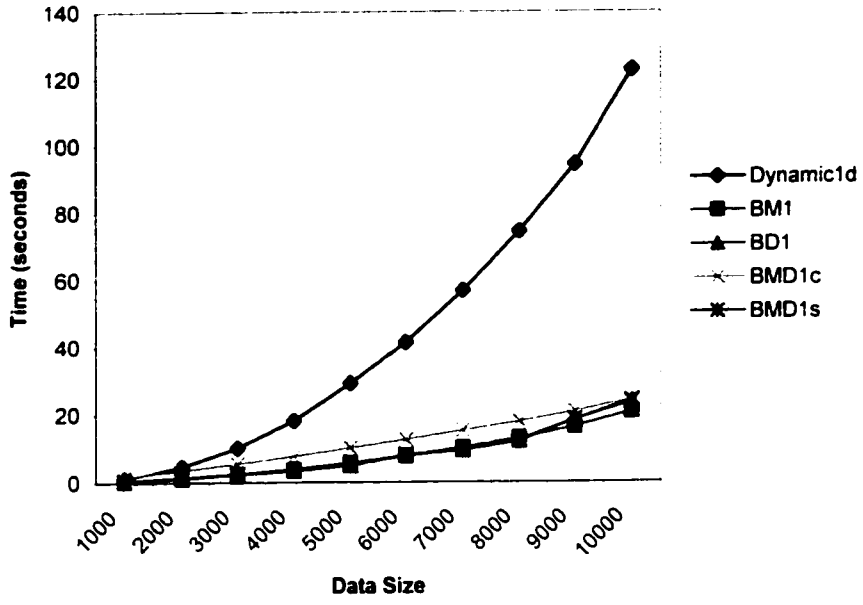

Figure 6.1 Times to Process Data of Various Sizes

Though dynamic1d will return the optimal solution, in the instance of a very large data set it might be desirable to approximate that solution by another method so an answer can be found in less time. From the table it seems that there is little difference with respect to time for the local search algorithms. Though BMD1c is consistently the slowest, for the largest set tested it is on average less than four seconds slower than the fastest algorithm. Also, the average time required for all these algorithms to find a solution increases at nearly the same rate. With this it seems the accuracy of each algorithm should be the deciding factor in choosing a local search method to approximate solutions.

The *error* for an algorithm will be defined as the difference between the value dynamic1d returns and the value that algorithm returns for the objective function for a particular data set

$$P(\Delta_{dynamic1d}) - P(\Delta_{local\_search}). \quad (1)$$

$\Delta_{algorithm\_name}$ is the partition or solution returned by a particular algorithm. The value given by (1) will always be non-negative since dynamic1d is guaranteed to return the optimal solution.

The same sets of vectors used to illustrate the average differences in times between the algorithms will be used to show the average errors for the algorithms. Table 6.3 contains the average errors for each algorithm with the data sets of size 100 and 1000 to 10,000.

Table 6.3 Average Error for Solutions to Thirty Data Sets

Data Size	Algorithm			
	BM1	BD1	BMD1c	BMD1s
100	5.90	0.37	5.84	0.37
1000	74.04	2.52	73.59	2.52
2000	152.76	4.00	151.78	4.00
3000	231.15	4.11	229.60	4.07
4000	299.62	8.01	298.02	8.01
5000	379.11	9.77	376.95	9.77
6000	447.88	11.01	445.44	11.01
7000	528.25	11.71	526.05	11.71
8000	603.79	13.01	600.98	13.01
9000	671.74	15.02	668.70	15.02
10000	770.41	15.10	765.00	15.10

As the size of the data grows, so does the error for each algorithm. BM1 and BMD1c both have the largest average error, BM1 being slightly greater. These also appear to increase at the same rate. Similarly, BD1 and BMD1s have the smallest error, actually equal for each of the data sets except for the sets of size 3000. In that case BMD1s found slightly better solutions than BD1 on two occasions, the other twenty-eight were equal. The change in error as the size of the data sets increases is shown in figure 6.2.

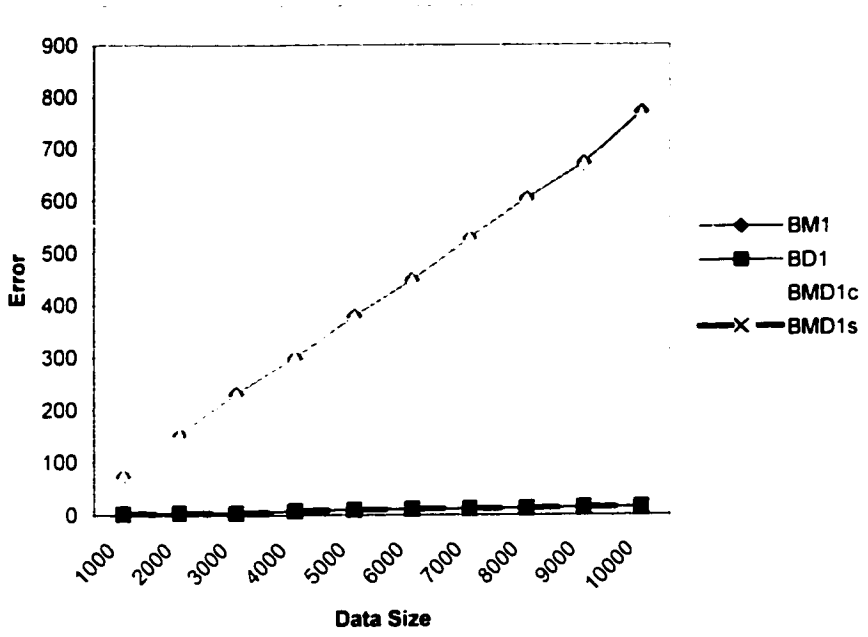

Figure 6.2 Changes in Errors as Size of Data Increases

The consistently better approximation of the optimal solution by the top-down methods is to be expected since the data being processed has very little structure. In data with many change points this pattern might be reversed, with the bottom-up methods returning better results. This is of little help in considering which local search method to use in approximating a solution if it is not known before processing whether the data has a great deal or only a small amount of structure. In the application that generated this optimization problem, data usually has relatively few change points. In that case, the best approximation by a local search would be by one of the top-down methods.

Another question to be answered is how many of the change points in the optimal solution are found by the local search algorithm. An interesting property of the objective function to notice, and possibly investigate further at some other time is that as the size of the data increases, and the rate does not, the optimal solution tends to have more change

points. The data is generated by finding a given number of non-repeated random values between 0 and some maximum value, so it is assumed to have very little structure. The process does not change as the size of the data increases, but more random structure, as defined by the objective function, does occur in larger data sets.

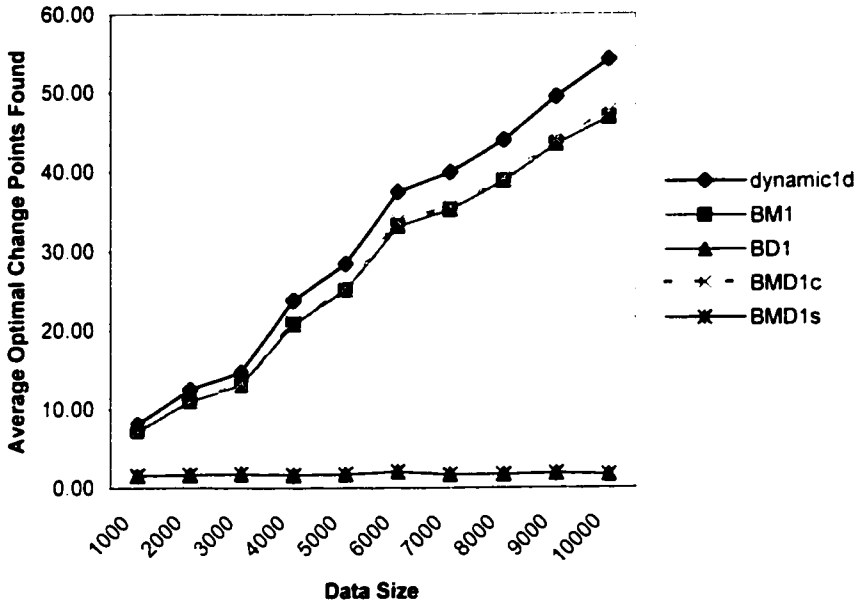

Figure 6.3 Average Optimal Change Points Found by each Algorithm

Figure 6.3 shows approximately how well each algorithm finds optimal change points. The plot for dynamic1d shows the average number of optimal change points for data of a given size. This clearly shows the increased random structure found in larger data sets. The bottom-up methods do not find all the change points, but do on average find most of them. On the other hand, both top-down algorithms consistently find about the same number of true change points.

So, though the average error is small for top-down methods, it appears they do not find small variations in the data. Bottom-up methods are able to find many of these

change points, but as will be shown, they suggest much more structure exists than truly does.

The following table shows the average number of false change points found by each algorithm. The increase seen in the bottom-up methods is linear, with an approximate rate of increase of 88 false change points for every 1000 data points.

Table 6.4 Number of False Change Points Found by Each Algorithm

Data Size	Algorithm			
	BM1	BD1	BMD1c	BMD1s
100	7.50	0.13	7.60	0.13
1000	82.20	0.13	82.77	0.13
2000	170.50	0.07	171.30	0.07
3000	260.77	0.40	262.50	0.37
4000	354.00	1.00	357.00	1.00
5000	434.90	0.13	436.87	0.13
6000	515.47	0.10	517.97	0.10
7000	600.57	0.20	604.57	0.20
8000	689.60	0.10	692.53	0.10
9000	770.63	0.20	775.47	0.20
10000	873.07	0.17	876.40	0.17

Here it is suggested that though BD1 and BMD1s both do not find many change points, those they do return are usually true change points. Also, as BM1 and BMD1s find many of the true change points, many false change points are returned as well. Though all the local search methods are very efficient in terms of time, they all have serious drawbacks. The bottom-up methods find too many change points and do not approximate the optimal value for the objective function well, and the bottom-up methods are unable to find the subtle structure contained in the data.

6.3 Approximation Algorithms

The two approximation algorithms, tabu search and simulated annealing, will be referred to as *tabu1d* and *SA1d*, respectively. A small portion of the parameter space for each algorithm is sampled, and the methods are evaluated on how those values affect performance. A more in-depth study of the parameters might yield better results, but the total time required to find an approximate solution to a given problem would then far exceed that required by the dynamic algorithm.

The amount of time to find a solution is mostly determined by each algorithm's input parameters. Good results follow only if these are chosen within some reasonable limit. Both utilize a fixed number as a stop criterion. This can be made smaller to decrease the time to find a solution, but if this number is too small, poor results will follow.

Tabu search simply requires that the stop criterion be such that the better solutions found by using the tabu list are achieved. Here the stop criterion was made sufficiently long to hopefully guarantee the best solution for a given length of the tabu list would be found. It is assumed that the time required will be considerably more than a local search algorithm. A more complete study of these values might show that a smaller number would suffice for the stop criterion, though not one that allows a solution to be found in less time, or fewer iterations, than a local search with a similar start state.

Another factor that can affect the time required for tabu search, but that cannot be accommodated by changing parameters, is that a record of the best solution so far, and the posterior value for that solution, must be kept and updated as the algorithm runs. The time required will increase if this is done frequently. For data that has few blocks in its

optimal partition a top-down approach would again be favored since the best solution values would need to be updated only a few times.

The time required for the simulated annealing algorithm to find a solution is heavily dependent on the stop criterion, C, and the length of the chains L_k, $k = 1,....,C$. The stop criterion should depend somewhat on the decrement constant α, and the control parameter c_0. As the algorithm performs actions in the last few chains very few, or none, of the non-beneficial actions should be performed. Since these are done with the probability given in (4.3), $e^{\frac{m \cdot df(h,l)}{c_i}}$, where $c_i = \alpha^k \cdot c_0$, should become small enough that such actions will almost certainly not be carried out. So, as the value for c_0 increases, or α approaches 1, C must also be increased.

As actions are picked at random, the L_k need to be sufficiently long so that two things may occur. First, early on in the algorithm, some non-beneficial merges and divides should be allowed to help move beyond some local maxima. Not too many of these should be allowed, since it is not desirable to put the system in a state that is far from the optimal one. The last few chains need to be long enough that most, hopefully all, beneficial actions will be carried out to achieve the highest objective function value possible. In the examples to follow, the chains are increased in length by 10 each cycle.

Table 6.5 Average Times (in seconds) for tabu1d to find a Solution on 1000 Data Points with Varying Tabu List Lengths

t	C	Bottom-up	Top-down
2	2000	91.13	268.37
4	2000	90.83	222.97
6	2000	91.18	205.41
8	2000	91.50	184.83
10	2000	91.56	174.51
12	2000	91.86	163.84

Table 6.6 Average Times (in seconds) for SA1d to Find a Solution on 1000 Data Points with Various Cooling Schedules

α	c_0	C	L_1	Bottom-up	Top-down
0.8	10	100	25	8.21	10.21
0.9	10	150	25	10.35	12.60
0.95	10	200	25	12.13	14.53
0.99	10	1000	25	31.24	33.84
0.8	20	100	25	8.58	10.70
0.9	20	150	25	10.91	13.30
0.95	20	200	25	12.42	14.78
0.99	20	1000	25	31.60	34.27
0.8	50	100	25	9.21	11.45
0.9	50	150	25	11.35	13.73
0.95	50	200	25	12.68	15.27
0.99	50	1000	25	32.28	34.62

Tables 6.5 and 6.6 illustrate the increase in time necessary to find solutions with the prescribed sets of parameters. For the same data, the dynamic algorithm took an average of 1.18 seconds to find a solution. As seen in (6.2), this increases considerably with an increase in the size of the data set. These algorithms should not see such an increase since they are generalized forms of the local searches. The parameters necessary to find reasonable approximations should cause only a linear increase in time with the size of the data.

For moderately sized one-dimensional data dynamic1d finds a solution in a reasonable amount of time, and is guaranteed to be optimal. So these approximations are not likely to be a desirable option given the times above. This is confirmed by the average errors given in tables 6.7 and 6.8.

Table 6.7 Average Errors Over 30 Vectors of Size 1000 for tabu1d

t	C	Bottom-up	Top-down
2	2000	65.26	3.80
4	2000	61.27	3.42
6	2000	57.93	5.59
8	2000	55.73	4.06
10	2000	54.18	6.46
12	2000	52.02	6.77

Table 6.8 Average Errors Over 30 Vectors of Size 1000 for SA1d

α	c_0	C	L_1	Bottom-up	Top-down
0.8	10	100	25	114.76	86.08
0.9	10	150	25	351.39	336.79
0.95	10	200	25	661.03	665.84
0.99	10	1000	25	782.43	766.51
0.8	20	100	25	146.92	134.44
0.9	20	150	25	552.76	550.98
0.95	20	200	25	767.53	760.67
0.99	20	1000	25	818.46	808.54
0.8	50	100	25	231.41	220.42
0.9	50	150	25	728.11	731.97
0.95	50	200	25	816.22	817.65
0.99	50	1000	25	835.56	839.62

Though this is a rather light analysis of these methods, it is interesting to note a few properties. In comparison to the earlier results of local search methods for 1000 data points, tabu search does seem to show better results with a bottom-up method, with a preference for longer tabu lists. The list length preference seems to be the opposite for the top-down approach. The errors however, do not seem to differ greatly from those given by similar local searches.

The errors given by SA1d are considerably higher than those seen for tabu1d. This may be due to the lack of a more complete analysis of potential cooling schedules, but further drives home the point that while the time required may increase linearly, a great deal more time will likely be invested in finding a suitable cooling schedule. While this may not be the only recourse here, in two and higher dimensions such an investment of time may prove to be worthwhile.

One note on the simulated annealing simulations cited above. Since it is a random process, different results might be given for the same set of data on multiple runs with the same cooling schedule. So the values above might not be the best solutions possible even for the cooling schedules used. This is not a great problem though, since the hope is to find a solution in a single run of an algorithm.

Table 6.9 Average Number of Missed Optimal Change Points for tabu1d

t	C	Bottom-up	Top-down
2	2000	3.97	0.93
4	2000	0.63	6.77
6	2000	0.63	7.13
8	2000	0.77	7.10
10	2000	0.90	6.67
12	2000	0.93	7.03

Table 6.10 Average Number of False Change Points for tabu1d

t	C	Bottom-up	Top-down
2	2000	76.00	1.47
4	2000	72.87	1.03
6	2000	70.67	2.60
8	2000	68.60	1.77
10	2000	67.70	3.00
12	2000	65.37	3.23

Tables 6.9 and 6.10 further show the effect of the tabu list length on the solution given by the algorithm. As seen in the local searches the bottom up versions found many change points not part of the optimal solution. Again, results tend to improve for this as the list length increases. The top-down tabu1d did seem to find few false change points on average, but more than the local search methods presented in (6.2).

Table 6.11 Average Number of Missed Optimal Change Points for SA1d

α	c_0	C	L_1	Bottom-up	Top-down
0.8	10	100	25	0.00	3.93
0.9	10	150	25	1.80	2.10
0.95	10	200	25	2.87	2.63
0.99	10	1000	25	4.07	3.60
0.8	20	100	25	3.23	3.40
0.9	20	150	25	2.13	2.33
0.95	20	200	25	3.40	3.70
0.99	20	1000	25	3.93	3.77
0.8	50	100	25	3.43	3.47
0.9	50	150	25	2.67	2.90
0.95	50	200	25	3.73	3.60
0.99	50	1000	25	4.00	3.53

Table 6.12 Average Number of False Change Points for SA1d

α	c_0	C	L_1	Bottom-up	Top-down
0.8	10	100	25	77.40	63.10
0.9	10	150	25	208.73	204.23
0.95	10	200	25	385.57	390.53
0.99	10	1000	25	453.20	445.47
0.8	20	100	25	95.80	87.80
0.9	20	150	25	322.17	320.80
0.95	20	200	25	447.70	443.53
0.99	20	1000	25	476.00	472.07
0.8	50	100	25	138.63	132.47
0.9	50	150	25	422.20	425.90
0.95	50	200	25	474.50	475.03
0.99	50	1000	25	488.10	489.37

Similar results as to what is given in Tables 6.9 and 6.10 are shown in Tables 6.11 and 6.12 for SA1d. In general, the performance of the algorithm here tends to deteriorate as the decrement constant approaches one, and as the initial temperature is raised. Starting with a single-block initial partition is not as advantageous when used with simulated annealing as was with tabu and local searches. In fact, there is little difference in the number of optimal and false change points found between the top-down and bottom-up approaches. Fine-tuning the cooling schedule would very likely lead to improved results but would increases the total time dedicated to a problem which could be solved fairly quickly by dynamic1d.

6.4 Structured Data

The performance of all algorithms when finding a partition for a data set that has a known structure will be compared. From what is given in (6.3) the objective function will define more structure than what is known to exist. Every set of purely random data had some structure. Ideally, a reasonable sub-optimal solution should find most of the change points given by dynamic1d, and only a few additional change points.

A single data set, norm_10000, was generated in much the same way as the sets tested in (6.2) and (6.3). Rather than having a single intensity through the entire data set, several sections of various intensities were generated. Since the data is generated randomly, the rates are only approximate. On each section though, the overall rate, or number of points divided by the length of the interval, is near the value stated. The block sizes and approximate rates are given in table 6.13.

Table 6.13 Rates and Sizes of the Known blocks in norm_10000

Rate	0.01	0.03	0.06	0.08	0.10	0.08	0.06	0.03	0.01
Size	1500	1250	1000	1000	500	1000	1000	1250	1500

Figure 6.4 is a histogram of the 10000 points of norm_10000 using 50 bins of equal length. This shows the increase in intensity to the central peak and decrease from that peak to the end of the data. Table 6.14 shows the performance of the local search algorithms on norm_10000. Again, the bottom-up methods find far too many change points, while the top-down methods find too few.

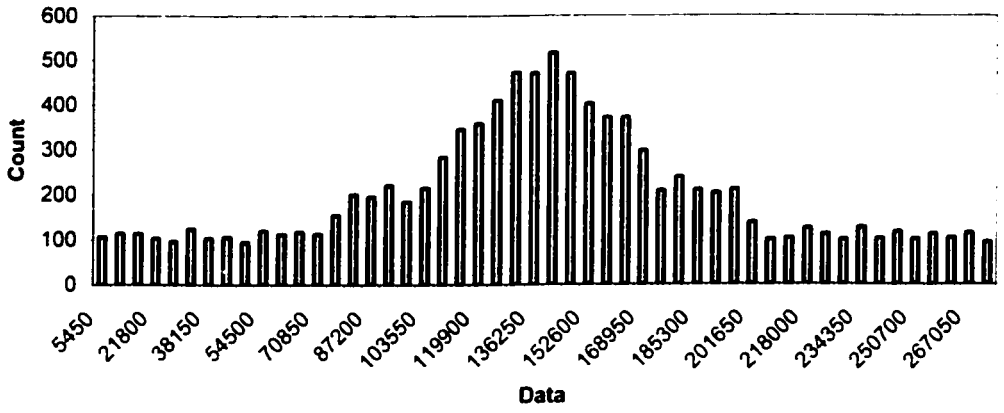

Figure 6.4 Histogram of norm_10000 with 50 bins

Table 6.14 Local Search Algorithm and dynamic1d results for norm_10000

	Time	Error	Optimal Change Points	Missed Change Points	False Change Points
dynamic1d	152.12	0.00	85	0	0
BM1	26.06	722.99	76	9	754
BD1	51.75	29.53	6	79	5
BMD1c	31.58	719.76	76	9	759
BMD1s	45.07	28.22	6	79	4

The following figures give a visual representation of the block structure for the data norm_10000 as defined by the stated algorithms. The plots show the length and intensity of each block. The vertical dashed lines show the location of the known change points. The result from dynamic1d is the optimal partition of the data so the structure defined by the other algorithms must be evaluated with respect to these blocks. The optimal partition of the data not only includes the change points that are known to exist, but also a number of very small blocks of high intensity which appear as spikes in figure 6.5.

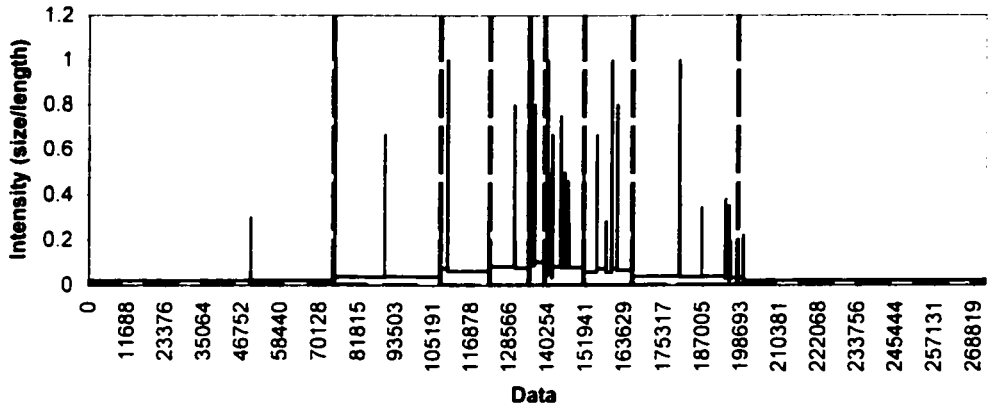

Figure 6.5 dynamic1d with Known Change Points

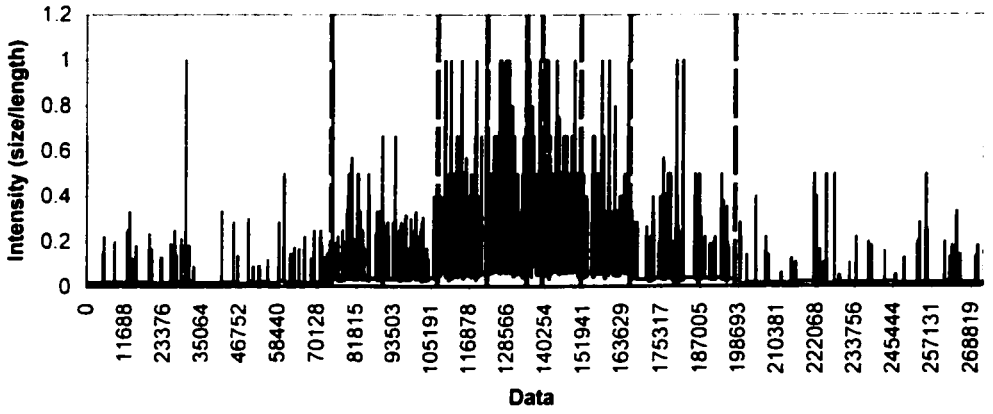

Figure 6.6 BMD1c with Known Change Points

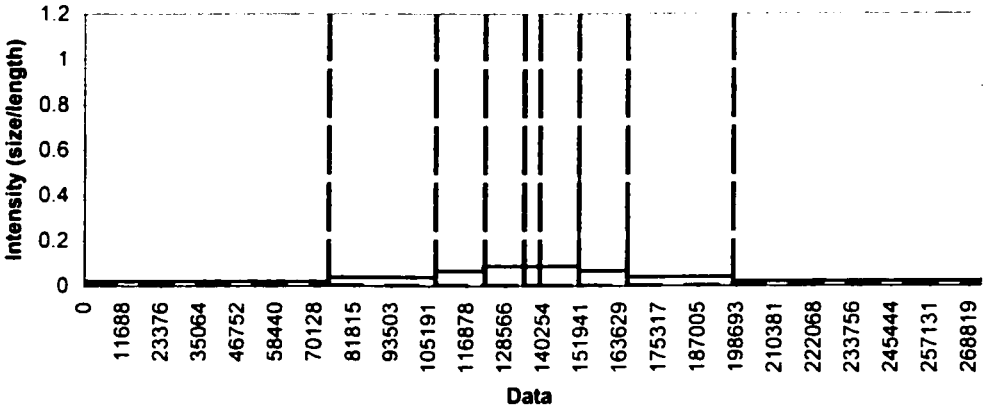

Figure 6.7 BMD1s with Known Change Points

Figures 6.6 and 6.7 show the best results for a top-down and bottom-up local search, respectively. BMD1c models the data with too many blocks, so though it does find many of the true change points, it is impossible to differentiate those from false change points without the knowledge of the true solution. BMD1s does the opposite. This algorithm finds most of the known change points, but is unable to find any of those that exist in the three regions at the central peak. Also, only two of the small blocks seen in the optimal solution are approximated. The change points found are also not all optimal, some seem to be close, but not exactly equal to the true boundaries of the block given in the optimal partition.

The following two tables give summaries of results for SA1d and tabu1d with varying parameters to find solutions for norm_10000. Simulated annealing shows little improvement over local search, and takes considerably more time to find a solution. Tabu search found some improvements on local searches with comparable starting states, but more time was necessary to achieve these results.

Table 6.15 SA1d Results with Various Cooling Schedules for norm_10000

	α	c_0	C	L_1	Time	Error	Optimal Change Points	Missed Change Points	False Change Points
dynamic1d	0	0	0	0	123.17	0.00	85	0	0
Bottom-up	0.8	10	500	25	139.57	5863.76	54	31	3018
	0.9	10	750	25	161.19	5856.38	46	39	3031
	0.95	10	1500	25	226.71	7419.90	43	42	3805
	0.99	10	2000	25	338.46	8664.76	35	50	4447
	0.8	20	500	25	145.97	6192.02	56	29	3187
	0.9	20	750	25	170.76	7301.73	45	40	3782
	0.95	20	1500	25	242.00	8438.89	49	36	4335
	0.99	20	2000	25	283.75	9299.89	47	38	4749
	0.8	50	500	25	153.12	6897.11	55	30	3549
	0.9	50	750	25	180.68	8727.72	43	42	4473
	0.95	50	1500	25	244.28	9237.38	35	50	4643
	0.99	50	2000	25	290.24	9593.95	37	48	4847
Top-down	0.8	10	500	25	191.93	2105.01	34	51	1076
	0.9	10	750	25	213.66	4693.16	43	42	2490
	0.95	10	1500	25	292.42	7153.39	52	33	3700
	0.99	10	2000	25	364.37	8741.87	41	44	4411
	0.8	20	500	25	210.26	3047.69	31	54	1516
	0.9	20	750	25	245.76	6588.75	41	44	3370
	0.95	20	1500	25	303.08	8636.41	40	45	4374
	0.99	20	2000	25	355.85	9116.15	43	42	4626
	0.8	50	500	25	195.02	4379.33	40	45	2222
	0.9	50	750	25	261.59	8374.38	42	43	4241
	0.95	50	1500	25	314.63	9234.23	40	45	4691
	0.99	50	2000	25	379.38	9400.76	52	33	4791

For SA1d, the top-down and bottom-up results are very similar. For a given cooling schedule, the starting state is not as heavy a predictor of how good a solution might be, as was the case for the local search algorithms. The structure in norm_10000 did not help improve the performance of the algorithm over the averages given in tables 6.7 to 6.12.

Table 6.16 tabu1d Results with Various Tabu List Lengths for norm_10000

	t	C	Time	Error	Optimal Change Points	Missed Change Points	False Change Points
dynamic1d	0	0	121.02	0.00	85	0	0
bottom-up	2	10000	34.64	719.76	77	8	759
	4	10000	34.81	713.06	76	9	753
	6	10000	35.87	711.23	76	9	751
	8	10000	34.97	711.24	76	9	750
	10	10000	35.67	690.31	76	9	730
	12	10000	35.80	690.32	77	8	735
top-down	2	10000	1028.33	28.22	6	79	4
	4	10000	1299.37	28.22	6	79	4
	6	10000	1771.97	28.51	5	80	5
	8	10000	3010.67	28.45	6	79	5
	10	10000	2871.48	28.45	6	79	5
	12	10000	1367.86	28.21	9	76	7

The tabu1d algorithm improves on local search results, but in some cases at a fairly high cost in the amount of time necessary to terminate. The bottom-up starts appear to have a decrease in error as the list length increases, but these still suffer from the problem of finding too many change points. The top-down methods make a fair improvement over local search, but take considerably longer to terminate. For the data set above, the top-down solutions were actually found fairly early in the run. The bottom-up methods however, found solutions fairly late. The time to finish may be decreased with fine-tuning, and finding a better approximation of when a solution will be found. With no prior knowledge of the data, this would be difficult to do in practice. If a number of similar sets were to be processed however, earlier runs could help in the modification of the stop criterion as well as the list length.

In one dimension, the dynamic algorithm performs reasonably well in terms of the time to partition a data set. If an approximation is desired in less time, a top-down local

search method is most likely the best option for data with only a few blocks. For data with a large number of blocks, a bottom-up local search is probably the best approach. Approximations would only be necessary for very large data sets, so for most applications the sub-optimal algorithms would not be used since they yield results so far from the true solution.

For one dimension, there may be better ways of approximating the optimal solution using variations on dynamic1d that are not guaranteed to find the best solution, but that will usually find one very close, if not the optimal one itself. An example would be to divide the data using a divide only algorithm, and then find the optimal partition of each of the subsets. This could divide the order of the optimal algorithm by a constant, and hence find an approximate solution in far less time for very large data sets.

Though approximations might be used rarely in one dimension, in two dimensions there is not at present an efficient way to find the optimal partition. Thus, the approximations are the best result possible given a limited time to find a solution. In chapter 7 the various techniques used will be compared and evaluated in a similar manner as was done here. Finding a fairly time-efficient algorithm that consistently yields higher values than those given by other methods for the objective function is the goal of the survey.

Chapter 7

Algorithm Performance Comparison in Two Dimensions

7.1 Results in Two Dimensions

The two dimensional merge/divide algorithms to be compared differ from their implementations in dimension one by a restricted definition of a division. There are a number of ways to split one block of two-dimensional Voronoi cells in two, so to make the implementation more efficient, a division was defined as removing a single cell from a block. Though this may seem limiting, in many cases it does allow for poorly chosen merges in simulated annealing and tabu search to be undone, while requiring a minimal amount of additional computation and memory.

Some difficulty arises in implementing this idea since removing a single cell may create more than two blocks, which is not allowed by the structure underlying all these algorithms. Accommodating the case where multiple blocks are created by a divide can prove to be very tedious even if it is allowed. So it is necessary, and more convenient, to disregard divisions in which the cell being removed corresponds to a cut vertex in the dual graph of the block β to which it belongs. This in itself is a fairly time-consuming endeavor, so the more easily recognized case of removing a cell corresponding to a cut vertex for the graph generated by only itself and its neighbors that are also contained in β is labeled as an invalid action. Some valid divisions are overlooked, such as removing a cell that corresponds to a vertex on a cycle, but this loss in performance is acceptable

when compared with the great increase in time necessary to handle these less frequent cases.

In the programs created to implement the algorithms, invalid one-cell removals were found in the following way. If a cell c is to be removed from a block β, the Laplacian matrix Q for the dual graph of the neighbors of c that also lie in β is generated. If the rank of Q is two less than the number of vertices in this graph, then the graph is disconnected and the action is not allowed to occur [Bi]. Finding the rank of this matrix in two dimensions is not very time consuming since a cell will usually have no more than ten to twelve neighbors, and most have fewer.

As a result of defining divisions in this way, bottom-up algorithms will be the only methods used in this chapter. The algorithms that will be compared are: local search best merge, *BM2*; local search best merge/divide, *BMD2*; simulated annealing, *SA2d*; and tabu search, *tabu2d*.

In dimension two it does not seem possible to find a provably optimal solution in a reasonable amount of time, so there is no known best objective function value or partition with which to compare results given by the various algorithms. In addition, the partition boundaries are not so well defined. The comparison of change point locations, or the boundaries of blocks, in one dimension is fairly straightforward. In two dimensions, this is not done so easily since two blocks may contain many of the same cells, but their boundaries might differ greatly. Evaluating the relationship between similar blocks from different partitions of a single data set then becomes quite subjective. Thus the

comparison of the performance of the algorithms in dimension two will be different than that made for the algorithms in dimension one.

Two factors will be used to evaluate the algorithms in this setting: the time required to find a solution, and the objective function value for a solution. As was stated earlier, the goodness of fit for the models is not an issue that will be addressed with any rigor. The problem at hand is one of finding the greatest possible value for the objective function in some rather short period of time.

Differences in time to find a solution will be shown in a manner similar to that done in chapter 6. Averages over a number of sets will be taken to illustrate changes in time due to greater number of data points being processed. Average objective function values will also be computed to show how the various solutions compare to one another.

Some examples of structured data will be presented so the block composition returned by the various algorithms can be compared. Also, changes in objective function values may be seen to correspond to a change in the partition of the data. Only one method for visualizing models of the two dimensional data will be used, but other possibilities will be mentioned.

Performance comparisons based on results given for structured data may be a bit more telling when comparing objective function values and time required. In unstructured data there exist a few small blocks, but it is largely made up of a single region of nearly uniform rate. In comparison to similar simulations in one dimension, far fewer small blocks of high intensity should be seen. Thus unstructured data presents a worst-case

scenario for a bottom-up method to find a solution. Structured data might give a more reasonable estimation of how these algorithms perform.

All the implementations of these algorithms work equally well, though more slowly, in higher dimensions. The code in the appendices that returns the area, size, and adjacency matrix inputs would need to be changed to process higher dimensional data. Rather than finding the areas of the voronoi polygons, volumes of n-dimensional voronoi cells need to be computed. This is rather easy, since the environment used in all the implementations provides a built-in function that does most of the work in finding those volumes.

7.2 A Parameter Search for the Approximation Algorithms

Before performing the actual performance comparison, preliminary examinations of the parameter spaces for both tabu search and simulated annealing were carried out. This was by no means a complete analysis of the possible inputs for these two algorithms, but it did give a much better approximation of which regions of the parameter spaces would return respectable results.

The following two cooling schedules, given as $\{c_0, C, L_1, \alpha, (L_{k+1} - L_k)\}$, will be used in the comparison of the performance of algorithms on unstructured data.

$$c1 = \{80, 1500, 4, 0.9, 0\}$$

$$c2 = \{3, 600, 200, 0.9, 1\}$$

These sets are very different, but represent regions of the parameter space that on average returned better results in small-scale simulations than most other inputs. These also

represent two possible approaches to using simulated annealing: c1 uses short unchanging chain lengths, and c2 uses chains that are longer and increase in size.

These are not assumed to be the best of all possible parameter sets, or even the best sets of those tested. Schedules c1 and c2 gave good results relative to other cooling schedules tested in the preliminary search, and will be used to represent how simulated annealing performs in terms of time on two-dimensional data.

In the examination of parameters for tabu search a preference for rather long list lengths was found. Increasing the stop criterion C above two times the size of the data improved performance, but no more improvement seemed to appear with the used of a value higher than 2.5 times the set size.

Two parameter sets will be adapted to estimate the performance of the tabu search algorithm. Both t and C are changed to better suit data sets of different sizes.

$$t1 = \{.65 \cdot (\text{data size}),\ 2.5 \cdot (\text{data size})\}$$

$$t2 = \{.75 \cdot (\text{data size}),\ 2.5 \cdot (\text{data size})\}$$

Though the names t1 and t2 will be used to describe different parameter sets, the exact set should be clear from the context.

Again, no assumption has been made that these represent the best possible results for tabu search. Rather, they serve as a lower bound to potential solutions since a thorough investigation of the parameter spaces would most certainly find sets for both simulated annealing and tabu search that would improve upon the results presented here.

7.3 A Time and Function Value Comparison on Unstructured Data

The focus of this section is to illustrate approximately how the time required by each algorithm increases with the size of the data. The objective function values will also be presented and discussed, but far less random structure exists in two-dimensional data sets than in one-dimensional sets of comparable size and intensity. Thus, the function values for this data may not be good indicators of performance for some applications.

Most of the data sets tested here are considerably smaller than those examined in the previous chapter. The time and memory required to process two-dimensional sets of that size on a large scale is beyond the scope of the machine used in the implementations, so the sizes of the sets were decreased by a factor of 10 from those used in chapter 6. The increases in time relative to the sizes of the data should still be apparent from this analysis though, since processing data is more time consuming in two dimensions than it is in one.

Figure 7.1 shows the average time to return a solution for each algorithm. The local search methods are, as expected, some of the fastest methods tested. The tabu searches both took nearly the same time, and though the parameters were increased linearly with the size of the data, the time to find a solution appears to grow in a non-linear manner. The cooling schedule c2 was consistently the slowest of all the algorithms.

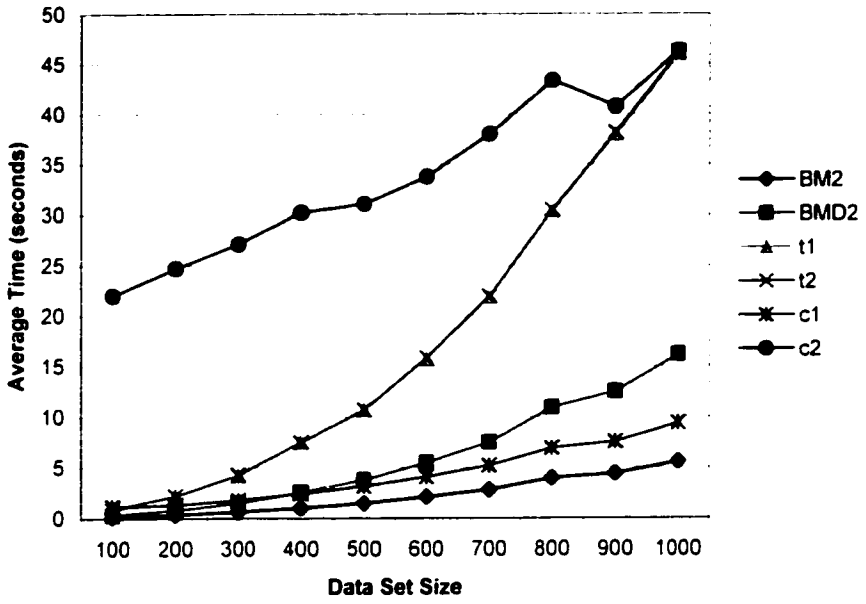

Figure 7.1 Average Times to Process Data of Various Sizes

Figure 7.2 illustrates the average rate of increase relative to the size of the data for this analysis. The local search algorithms both show a very slow rate of increase. Both tabu search and simulated annealing show growth as well, so none of the algorithms are of linear order. The less consistent growth in time relative to the size of the data for the versions of simulated annealing is probably due to the random nature of the algorithm itself, and the fact that the input parameters were not changed for data of different sizes. As a note, the plots for t1 and t2 are nearly identical. Upon close inspection however, the different symbols used to indicate the average values for given set sizes can be distinguished.

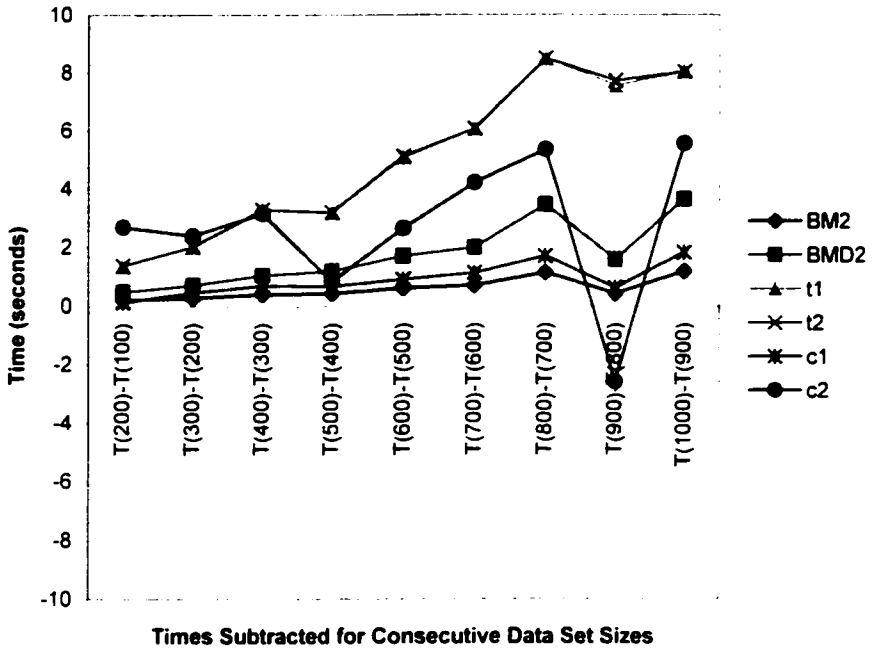

Figure 7.2 Differences in Average Times for Consecutive Set Sizes

With this it is apparent that in dimension two all the algorithms work in polynomial time greater than linear. BM2 seems to be of the lowest order, followed by simulated annealing c1 and BMD2, tabu search t1, t2, and simulated annealing c2, roughly in that order. The tabu searches really show no consistent difference in time between them. Just as in the last figure close inspection of the plot is necessary to find the slight differences between t1 and t2. The objective function values for these simulations will be shown to illustrate that the parameter sets chosen do return reasonable results.

Table 7.1 Average Objective Function Values for Each Data Set Size

Data Set Size	BM2	BMD2	t1	t2	c1	c2
100	-433.63	-433.63	-434.07	-434.07	-480.05	-474.59
200	-805.44	-805.44	-805.50	-805.50	-856.02	-843.44
300	-1177.48	-1177.48	-1177.53	-1177.53	-1233.38	-1231.23
400	-1547.62	-1547.62	-1547.65	-1547.65	-1605.65	-1612.46
500	-1927.58	-1927.58	-1927.60	-1927.60	-1996.37	-2006.75
600	-2293.24	-2293.24	-2293.27	-2293.27	-2363.09	-2389.31
700	-2661.31	-2661.24	-2661.25	-2661.25	-2732.60	-2753.89
800	-3026.26	-3026.05	-3026.08	-3026.08	-3108.78	-3128.39
900	-3397.37	-3397.32	-3397.33	-3397.33	-3479.12	-3510.74
1000	-3766.56	-3766.56	-3766.58	-3766.58	-3855.07	-3883.55

Table 7.1 shows both list lengths for the tabu search on average finding nearly the same solutions as the local search merge/divide algorithm. No algorithm ever improved on the best local search results. There was some variation in the values returned by BM2 and BMD2, but never any great difference. In only two set sizes, 700 and 800 is there any noticeable difference in the average function value returned for the local searches.

In many cases the tabu search algorithms found the same solution as the better of the two local searches. The simulated annealing algorithms, c1 and c2, never found a solution with a function value as high as either BM2 or BMD2, but their average function values do not seem too far from those given by the other algorithms. Figure 7.3 better illustrates how close on average all the objective function values are for each set size.

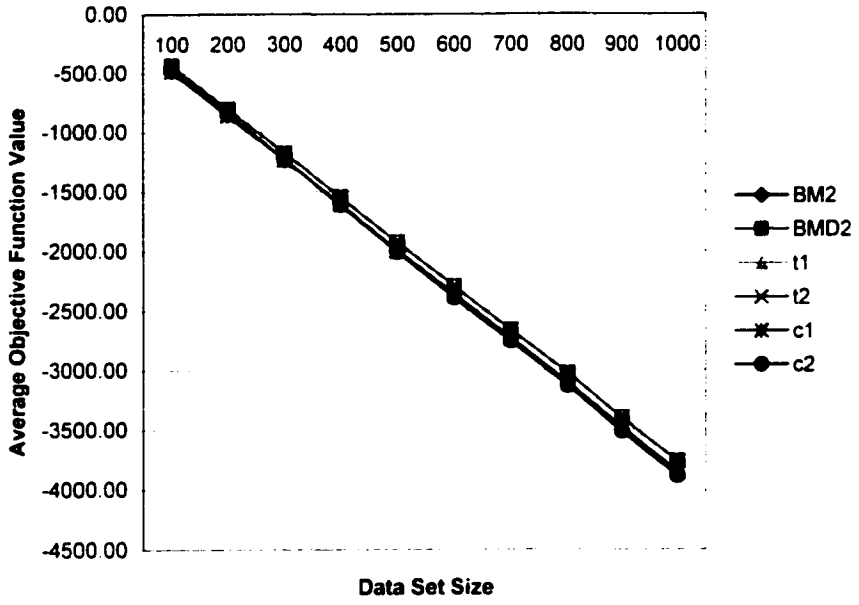

Figure 7.3 Average Objective Function Value Returned by Various Algorithms for Each Set Size

The average difference between the function value given by the better performing of the two local searches and each approximation algorithm is shown in Table 7.2. The tabu searches show no apparent growth with the size of the data. As the data set sizes increase, the difference between simulated annealing solutions and local search results, however, on average becomes greater. Figure 7.3 shows that the average function value in general becomes larger in absolute value when the set size becomes greater. The increased differences between simulated annealing and local search is likely due to these larger function values, and not decreased performance by c1 and c2.

Table 7.2 Average Differences in Objective Function Value for Approximation Algorithms and the Best Performing Local Search

Data Set Size	t1	t2	c1	c2
100	0.43868	0.43868	46.4205	40.9627
200	0.05472	0.05472	50.3174	37.4786
300	0.05019	0.05019	55.9032	53.7477
400	0.03268	0.03268	58.024	64.8395
500	0.01877	0.01877	68.7882	79.1672
600	0.03864	0.03864	69.85	96.0734
700	0.012	0.012	71.3543	92.65
800	0.02367	0.02367	82.7218	102.338
900	0.00911	0.00911	81.7982	113.42
1000	0.02105	0.02105	88.5192	116.995

The objective function values for the various algorithms will be investigated more thoroughly in the next section when these methods are used to process data with a known structure. A more clear distinction between large groups of points may provide a venue in which the approximation algorithms could perform better.

7.4 Structured Data

Six sets of simulated data were generated to evaluate the performance of the algorithms in such a setting. Each of these consists of three circular regions of high intensity occurring in a rectangular space, B, that is 400 units by 400 units containing random low intensity data, or noise. Each high intensity region, or cluster, contains approximately 250 points within a circle of radius $r = 30$ units. To generate each point in a cluster an angle was found simply by choosing a random number between 0 and 1 and multiplying it by 2π. The radius was generated in a slightly different way. A

random number r was again found between 0 and 1, and then this number was substituted in the following equation for a radius.

$$radius = 30 \cdot (1 - r^{2.35}) \tag{1}$$

The above equation gives a preference for greater radii to be chosen, while not generating too many points near the maximum radius allowed. With this regions of nearly uniform distribution are created. Equation (1) leads to near uniform distributions of points, but slightly more points than desired are generated near the centers of the circles. Increasing the power 2.35 remedies this, but causes a more noticeable problem with a preference to place points on or very near the boundary.

The sets differ by the placement of the high intensity regions. In each one the centers of the circular clusters of data are all placed a specified distance d from the center, (200,200), of the rectangular space with an angle between each of $\frac{2\pi}{3}$. The data sets will be referred to as set1, set2, and so on to set6. The distances from the centers of the clusters to the center of B are 60, 50, 40, 30, 20, and 10 for set1 to set6, respectively. Collectively these will be referred to as Group6.

The reason for generating data in this way was so that each algorithm processed several types of clusters and groups of clusters. When the circular regions are all rather far away from the center of B there are four regions of uniform intensity to be found by the algorithms: the three clusters and the background noise. As the circular regions become close to one another, the clusters may become less distinct. Finally, as the clusters begin to overlap, regions of higher intensity are created. Also to be expected

from what was seen in chapter 6, random structure should occur throughout the data. Figure 7.4 shows plots of each of the sets from Group6.

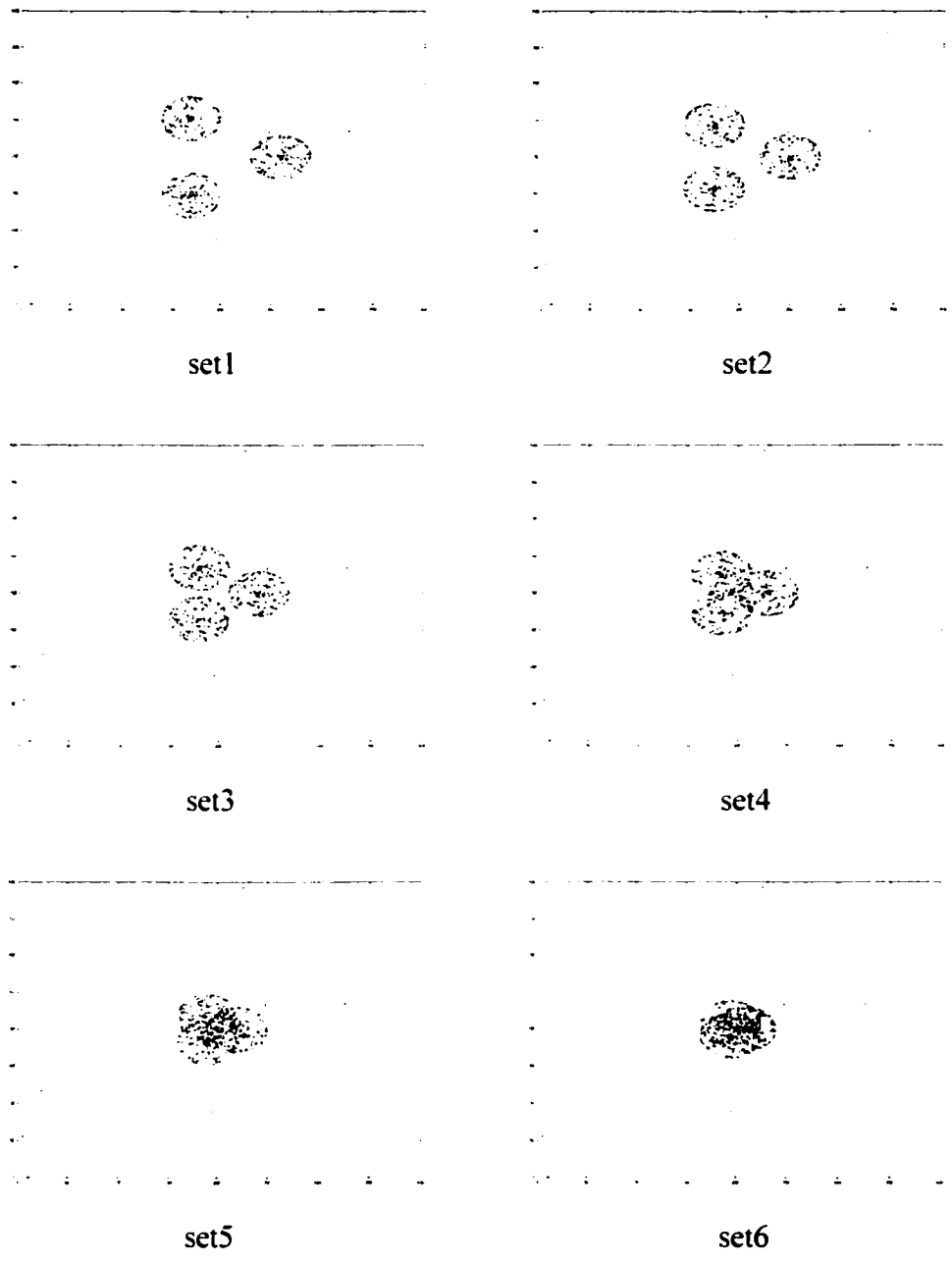

Figure 7.4 Structured Data Sets to Be Processed

Each algorithm presented in (7.3) was tested on all the data sets. Time, objective function value, and number of blocks in the model returned were used to compare the output of each method. There were some differences in the methods used in finding a solution to represent the performance of each algorithm.

The local searches, BM2 and BMD2, were both run once each, since no variation in results would come from running the algorithms multiple times or manipulating input parameters. Though only the list lengths from (7.3) are presented here, other list lengths were checked to verify t1 and t2 were returning good results. Other than the two mentioned, lists of length 10, and 100 to 1000 in increments of 100 were all tested. For every data set, the changes in length did not affect the solution found. Only the cooling schedules c1 and c2 were tested for simulated annealing. Some other schedules were tried on these sets, but c1 and c2 performed equally well or better than these. To obtain better results, each of c1 and c2 was run fifteen times on every set. Of these possible models, the one with the highest objective function value was used to show the potential performance of the parameters.

To display the partitions, all of the Voronoi polygons that are members of the same block are filled with shading of the same scale. Regions are filled based on the average intensity of the block. With this adjacent blocks are sometimes similar in appearance and less easily distinguished. Other methods might overcome this difficulty by utilizing visualizations that plot the intensity of each block in a third dimension. In each display the region most closely related to the background noise is filled with white to give the appearance of the blocks of greater intensity being filtered out from the noise.

It is interesting to note that some of the algorithms find solutions with the same number of blocks, but different objective function values. In all these sets, the best solution given by any of the algorithms was always either found by BM2 or BMD2. Similar to the results of (7.3), the tabu searches often find that same solution, or one very close to it. The simulated annealing trials found solutions close to those given by local search, but do not perform as well as tabu search. The limitations placed on the divide actions are likely to blame for this. Since simulated annealing allows non-beneficial actions to occur early on, later in the run it is necessary to undo many of these. Since the process is random, often blocks whose dual graphs are paths or cycles are generated in the first stages. With the divisions allowed these are very difficult to deconstruct, so they often result in poorly configured models. Many times when the real power of the algorithm is used to any great degree, such a situation occurs. Hence, the results for simulated annealing are expected to fall short of those given by local search.

Though six sets were tested, only three distinct cases occurred in comparing performance. The first was that BM2, BMD2, t1, and t2 all returned the same result. Second, BM2 and BMD2 found a better partition than t1 and t2. Finally, BMD2, t1, and t2 all improved noticeably over BM2. In all of these c1 and c2 each returned unique solutions with smaller objective function values than local or tabu searches. To minimize repetition of similar outcomes, only three of the six sets will be presented. These data sets, set1, set4, and set6, correspond in order to the cases noted above. The cases were replicated, again in order, by set2 set6, and set5, so there does not seem to be an obvious

correlation between the known structure of the sets and the type of results given by the algorithms.

The tables preceding the visualizations of the models summarize the results for all the algorithms. Since in many cases several algorithms return exactly the same model, only displays for distinct models will be given.

Table 7.3 Results for set1

Algorithm	Time	Objective Function Value	Number of Blocks
BM2	4.376	-4620.40181	47
BMD2	9.855	-4620.40181	47
t1	25.537	-4620.40181	47
t2	25.627	-4620.40181	47
c1	10.685	-4690.823044	46
c2	41.27	-4684.925432	49

The best partition found was given by BM2, BMD2, t1, and t2. This model is shown in figure 7.5. The three circular regions are fairly well defined, though the boundaries have some variation due to boundary points having relatively large voronoi cells. This causes a single block to be created connecting two of the clusters. The regions of high intensity in the centers of each circular area are due to the manner in which the data was generated.

Figures 7.6 and 7.7 show the results for the two simulated annealing cooling schedules used. The better objective function value of these two is given by c2. That model has less well-defined boundaries for the blocks, and includes a block that connects the three regions of higher intensity. Similarly in c1, the boundaries are not so well defined as the model returned by the local and tabu search algorithms.

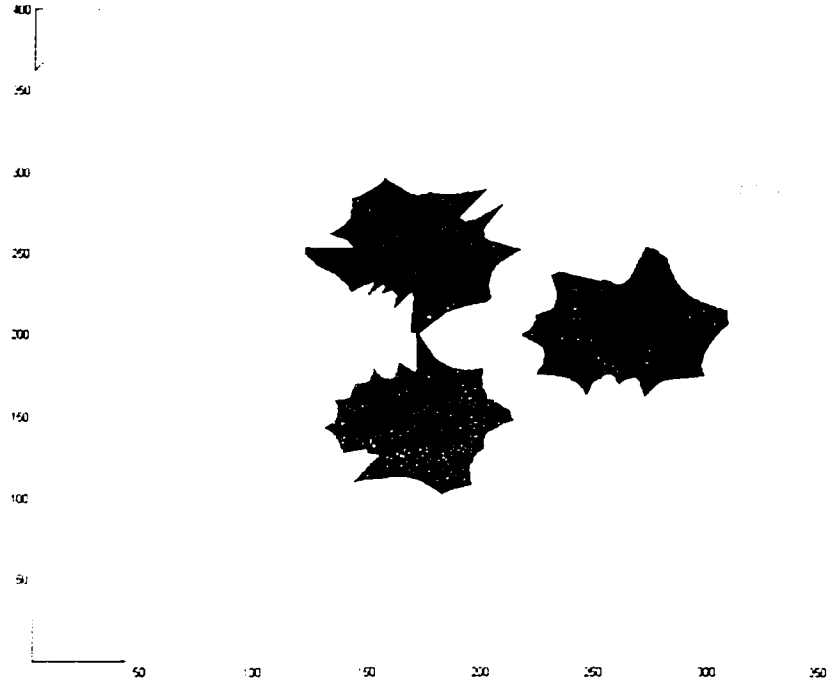

Figure 7.5 Model for set1 Given by BM2, BMD2, t1 and t2

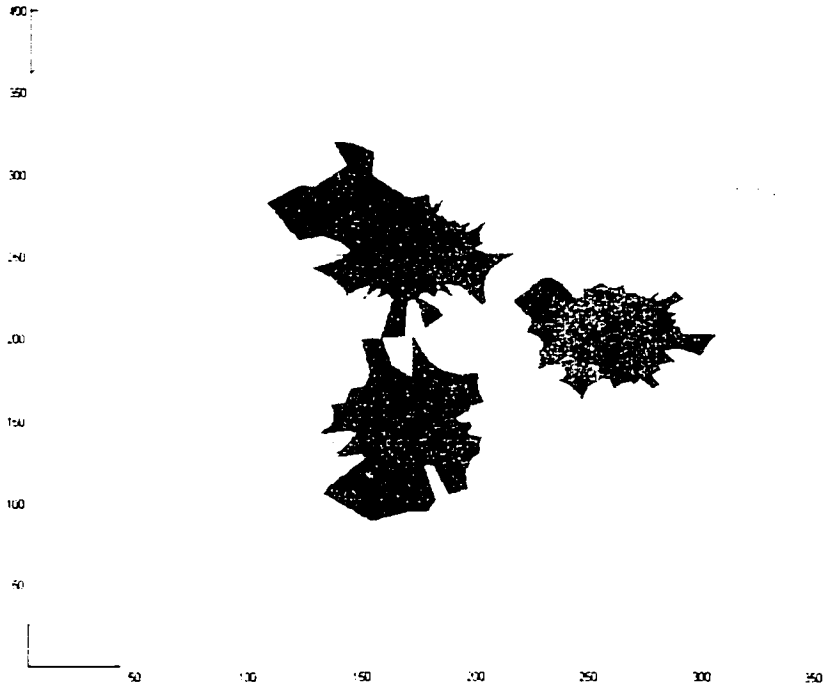

Figure 7.6 Model for set1 Given by c1

Figure 7.7 Model for set1 Given by c2

In set4 the tabu searches do not perform as well as the local search methods. A very small change in the model causes a slight difference in the objective function values. This change amounts to an additional small block returned by the tabu search. The difference can be seen in figures 7.8 and 7.9 in the regions highlighted by rectangles imposed over the plot. This example illustrates the degree to which a slight variation in structure affects the objective function.

Table 7.4 Results for set4

Algorithm	Time	Objective Function Value	Number of Blocks
BM2	4.506	-4455.811022	53
BMD2	10.324	-4455.811022	53
t1	24.896	-4455.820346	54
t2	26.098	-4455.820346	54
c1	12.649	-4492.434219	49
c2	46.326	-4495.343482	54

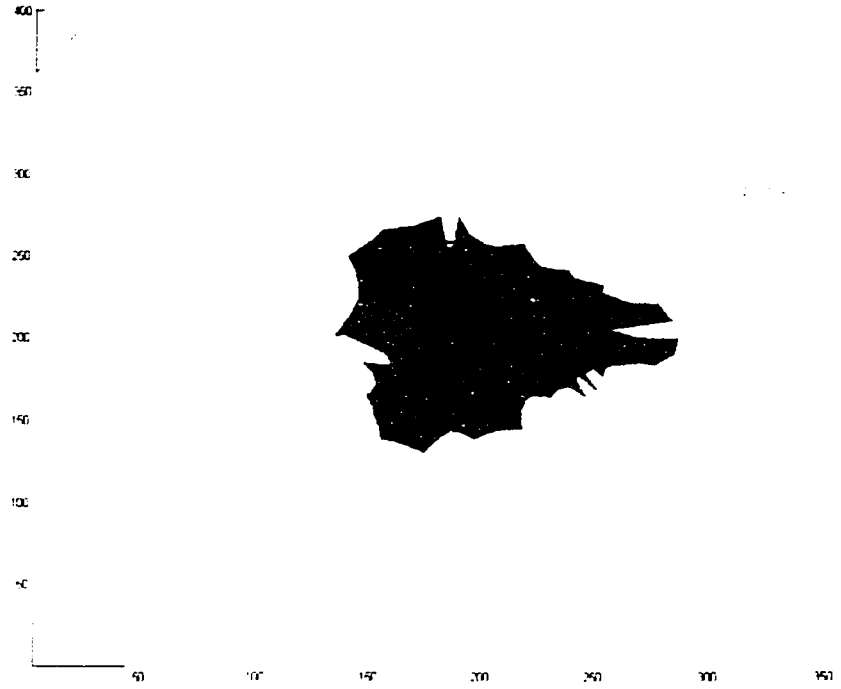

Figure 7.8 Model for set4 Given by BM2 and BMD2

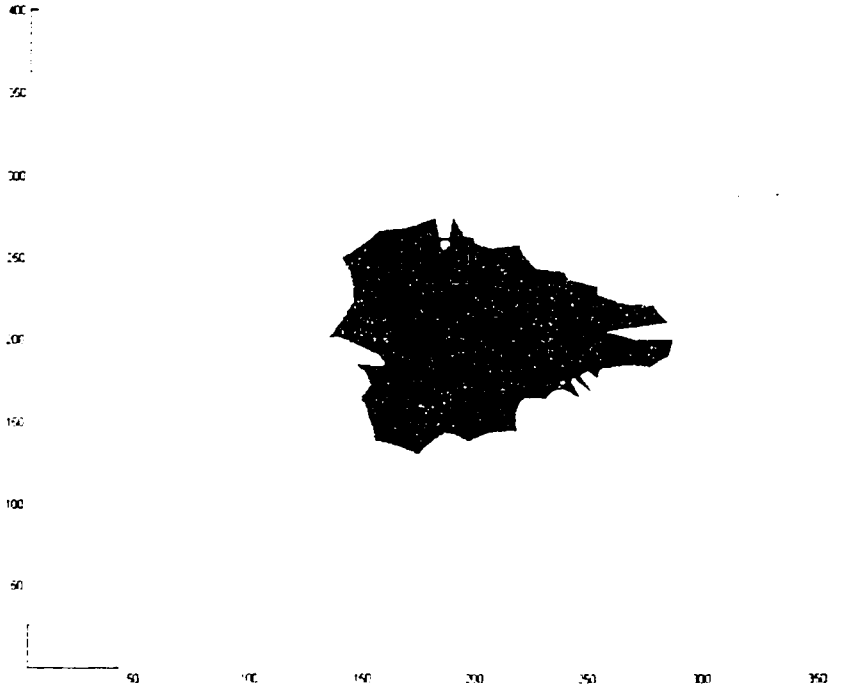

Figure 7.9 Model for set4 Given by t1 and t2

The cooling schedules c1 and c2 again generate models with boundaries that are quite different than those for the local and tabu searches, but simulated annealing generated models as a whole are much more similar to local search results than those for set1. The two models used to illustrate c1 and c2 differ primarily in the makeup of the larger regions. There are many similarities in the small high intensity blocks for all of c1, c2, the tabu and local searches. In addition, when generating all the potential models for c1 and c2, most of these also contained many of these same small blocks. The larger regions varied quite a bit over these twenty-eight other block structures. This included changes not only in the boundaries, but also in the larger partitions of interior regions. Figures 7.10 and 7.11 detail the simulated annealing partitions of the data.

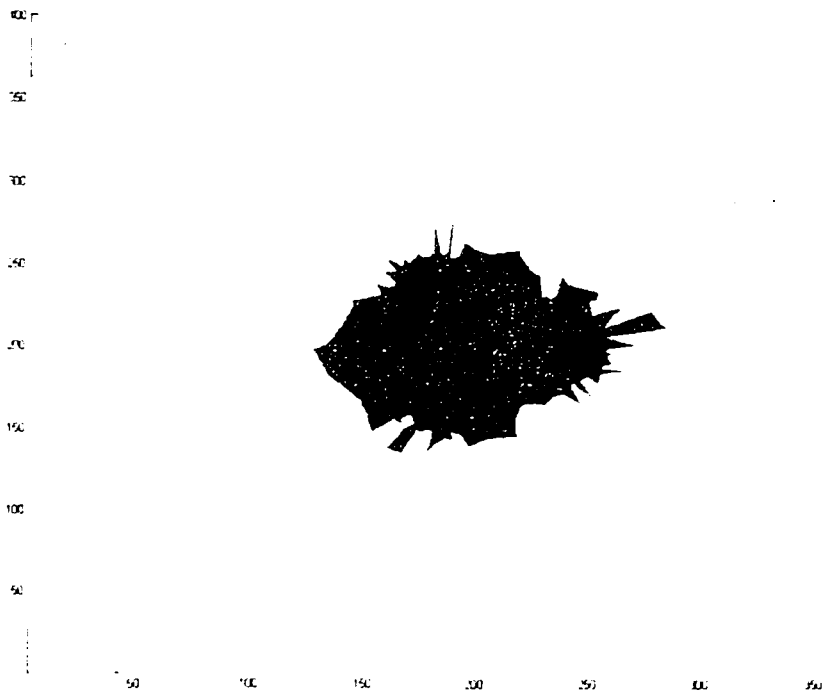

Figure 7.10 Model for set4 Given by c1

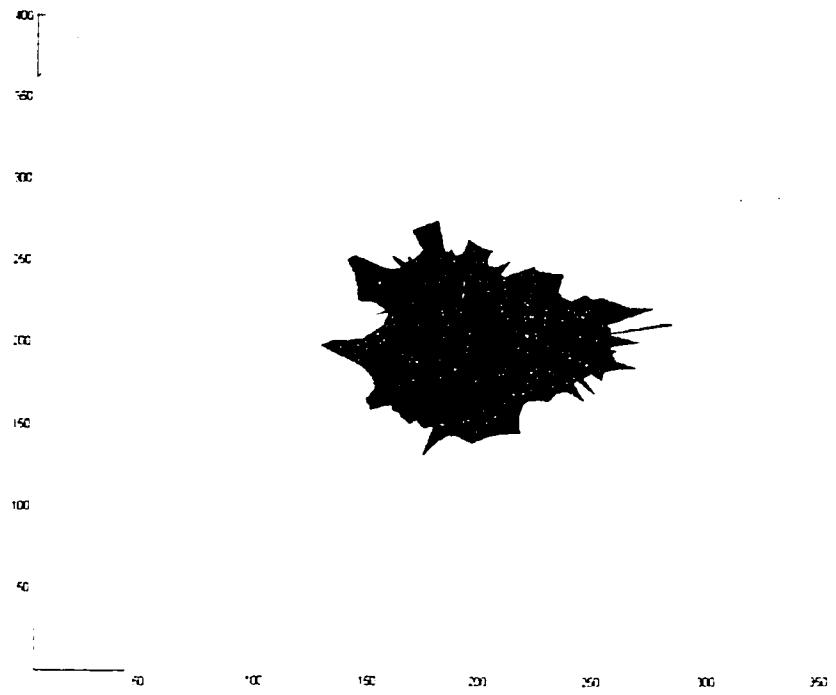

Figure 7.11 Model for set4 Given by c2

In the last set to be shown t1 and t2 again find the same solution as BMD2. In this case, however, that partition is better in terms of objective function value than that found by BM2. Though these two models contain the same number of blocks, the result given by BMD2 is nearly a three unit improvement over the model given by BM2 in objective function value. The most noticeable difference is a long narrow cell on the boundary on the upper left of the central region being associated with that central block by BM2 and with the background by BMD2. Also, some small blocks almost directly to the right of that cell differ between the two models.

Both solutions found with simulated annealing again vary quite a bit in the boundary of the central block, and the structure of the larger regions. Quite a few of the blocks with small areas in the interior of the central region found by both local and tabu searches are again returned by both c1 and c2.

Table 7.5 Results for set6

Algorithm	Time	Objective Function Value	Number of Blocks
BM2	4.487	-3927.509244	67
BMD2	10.305	-3924.66256	67
t1	29.503	-3924.66256	67
t2	30.013	-3924.66256	67
c1	13.059	-3981.739034	60
c2	51.253	-3982.133024	63

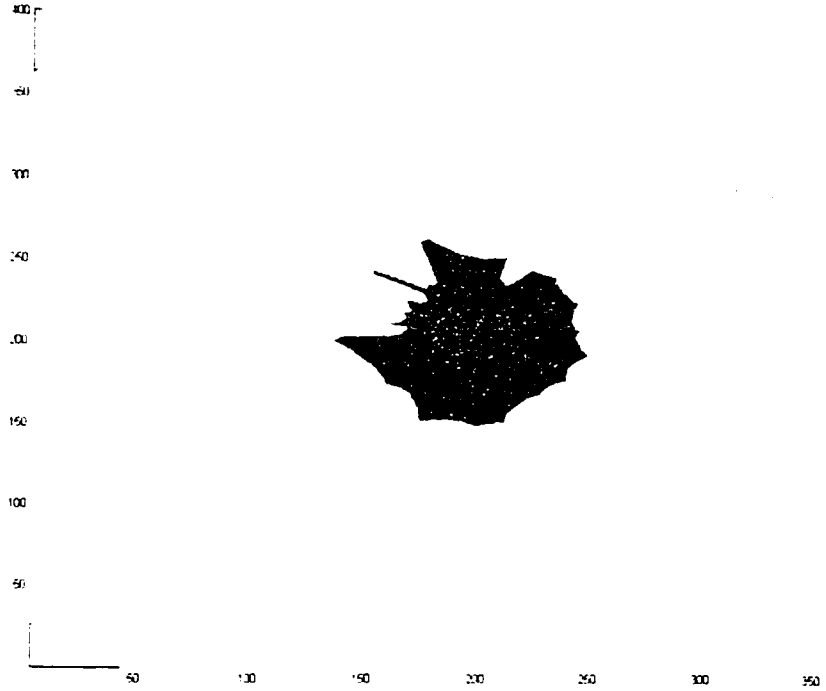

Figure 7.12 Model for set6 Given by BM2

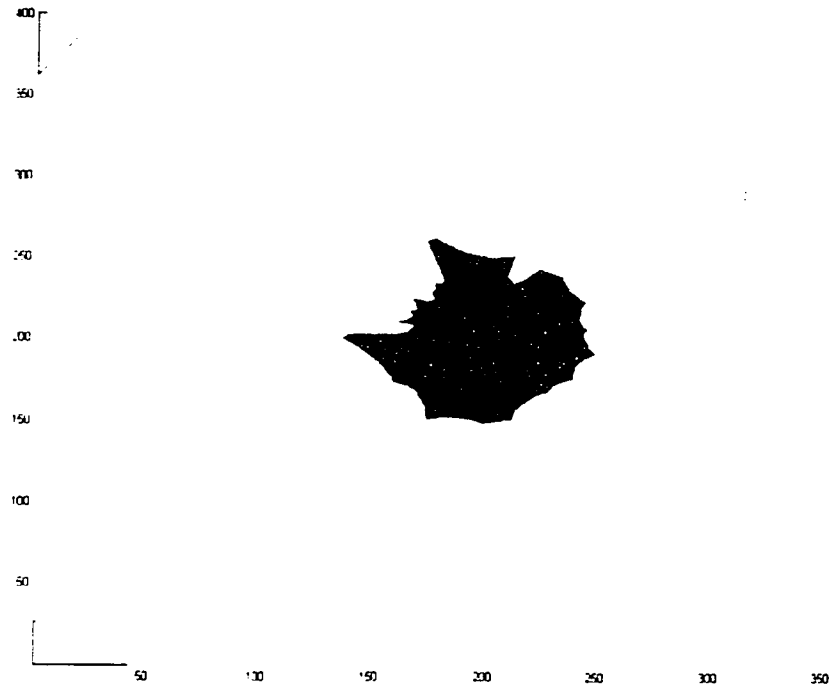

Figure 7.13 Model for set6 Given by BMD2, t1, t2

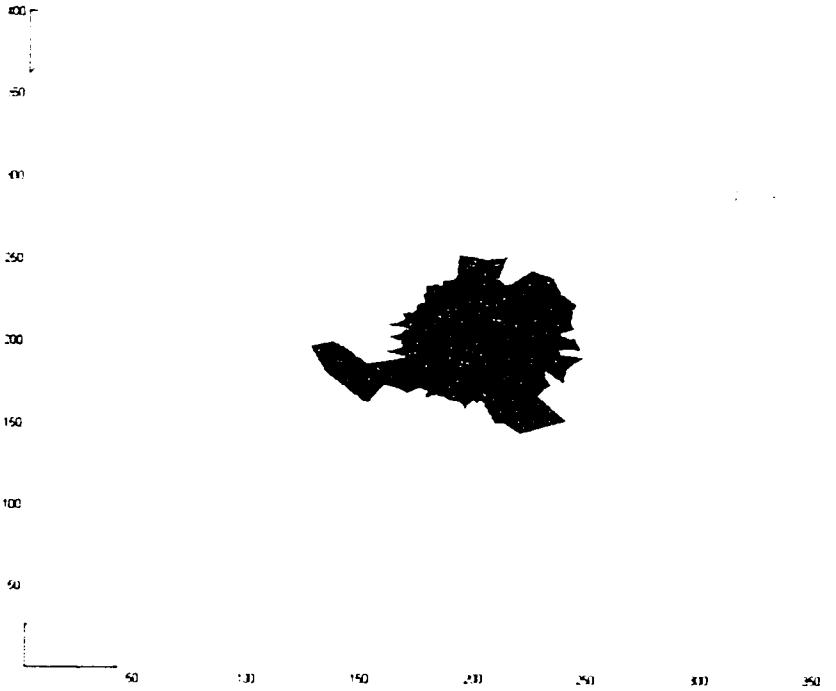

Figure 7.14 Model for set6 Given by c1

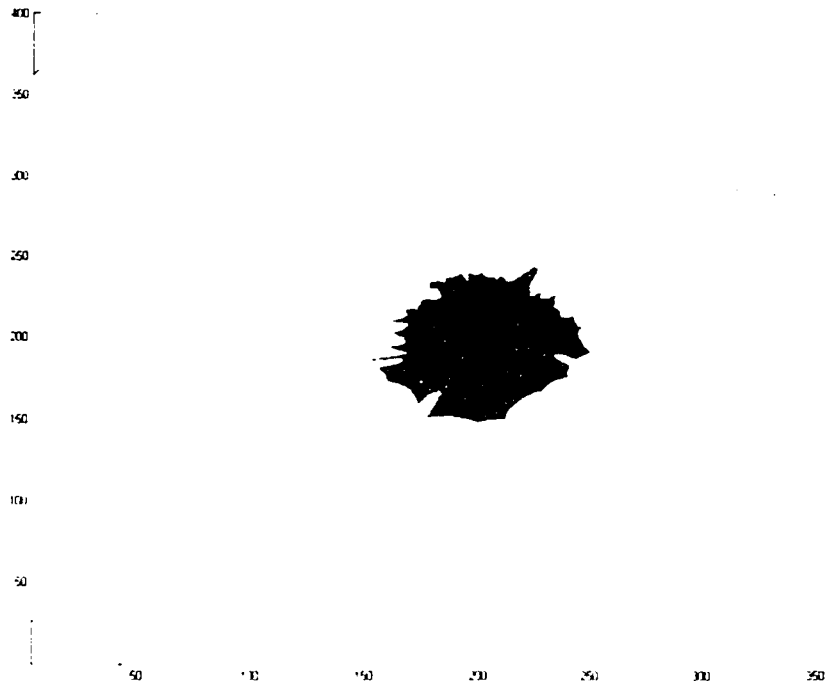

Figure 7.15 Model for set6 Given by c2

For the two dimensional problem, local search methods return the best results given the restrictions on the algorithms here. Tabu search usually finds solutions that are equally good, but does so in a greater amount of time. The cooling schedules chosen to represent the performance of simulated annealing gave reasonably good results, but like tabu search, did not make any improvement over local search in these examples. One of c1 or c2 did not consistently improve over the other, so there may exist many diverse cooling schedules which all return good solutions.

Results in general could be improved by adopting a more broad definition of a division, though a greater amount of time would be required by all of the merge/divide algorithms. Tabu search and simulated annealing would then likely yield better results with the proper choice of divisions, and parameter sets to accommodate such actions. Implementing this change would not be difficult, as the code in the appendices is

structured so that only minimal revisions would be necessary once a program that performs the prescribed action is written.

These and any new techniques based on this structure can be explored in three and higher dimensions. All of two dimensional function programs included in the appendices will work with higher dimensional data. The function that finds the areas and sizes of the Voronoi cells would have to be changed to find the volumes of higher dimensional cells. Also, the dual graph of the tessellation would have to be found in a slightly different way.

The method in general appears to work well based on these results and the example of structured data given in (6.4). If the optimal value for the objective function is found, then the corresponding model does appear to parallel the known structure of the data, with some additional random fluctuations. Even in the two-dimensional case, where the optimal value of the objective function is not known, the structures found are very close to what is expected. Further work will likely improve on the results here, and carry those improvements to higher dimensions.

Appendix

Implementations of the Algorithms

All of the programs were written and run in MATLAB®, © The Mathworks, Inc 2001. MATLAB is a matrix-based mathematical software package that allows for easy manipulation of vectors, matrices, and arrays in general. The function programs for all the algorithms utilize the built-in features provided by this environment. Vectorization, which amounts to replacing loops with vector operations or built in assignment and indexing commands, greatly improves performance. Whenever possible, these methods were used to make the algorithms more efficient.

Any line begun with a `%` is a comment line and is not part of the working code. Three dots: `...` are interpreted as a continuation symbol and when placed at the end of a line signify it continues on the following line. Any function called in these programs not included in the appendices is part of the Matlab library and documentation may be found in a Matlab reference text that covers release 12. For the following functions to run, they are saved in a directory recognized by the Matlab environment, each in a file with a name matching the function name followed by a .m extension. For example, if a function declaration is given as: `[output] = function_name(input)`, then that function should be saved in a file named `function_name.m`. The details for all these programs are contained in the comment headers just below the function declaration, and in comments within the code.

Appendix A Algorithms for One-Dimensional Data

This section contains all the function programs for the analysis of one-dimensional data. The first six sections, A.1 to A.6 are the function programs for the six algorithms tested: BM1, BD1, BMD1, tabu1d, SA1d, and dynamic1d. The final section contains all the programs necessary to support these algorithms, and one program, plotblocks.m, to visualize results.

A.1 Local Search Best Merge BM1.m

```
function [time, CPTs, posterior, heights] = ...
    BM1(length_vec, pop_vec)
%-------------------------------------------------------------
%BM1b_2    David Barnes    6/22/02
%   [time, CPTs, posterior, heights] = ...
%       BM1(length_vec, pop_vec)
%
%   A merge only local search algorithm for one-dimensional
%   data.  The two cells with the largest merge factor are
%   combined, then the process repeats until no two cells
%   have a merge factor greater than 0, or all the data
%   has been merged to one block.
%
%   input:  length_vec - lengths of the voronoi cells
%           pop_vec - lists the population of each cell
%
%   output:    time - the time to find a solution
%           CPTs - a vector of change point times
%           heights - a vector listing the intensities
%                     of each block
%              posterior - the objective function value for
%                       the model generated by the algorithm
%-------------------------------------------------------------
tic

nCells = length(length_vec);

% initial calculation of merge factors
mergeFactors = mrgfctr1(length_vec(1:nCells-1), ...
    pop_vec(1:nCells-1), length_vec(2:nCells), ...
    pop_vec(2:nCells));

maxmerge=1;
```

```
pass = 0;
%------------------- the merge algorithm ------------------
while (nCells>1&maxmerge>0)

    if(pass) % recalculate merge factors affected by merging
        L=length(mergeFactors);
        if (M<L)
            mergeFactors(M) = mrgfctr1(length_vec(M), ...
                pop_vec(M), length_vec(M+1) ,pop_vec(M+1));
            mergeFactors(:,M+1) = [];
        else
            mergeFactors(:,M) = [];
        end
        if M>1
            mergeFactors(M-1) = mrgfctr1(length_vec(M-1),...
                pop_vec(M-1), length_vec(M), pop_vec(M));
        end
    end

    % find the largest merge factor
    % return its value and index M in mergeFactors
    [maxmerge M] = max(mergeFactors);

    % merge the two blocks with largest merge factor (if >0)
    % to block with smallest in length_vec & pop_vec reset
    % to reflect this change
    if (maxmerge>0)
        length_vec(M) = length_vec(M) + length_vec(M+1);
        pop_vec(M) = pop_vec(M) + pop_vec(M+1);
        length_vec(:,M+1) = [];
        pop_vec(:,M+1) = [];
        % decrease the size of length_vec & pop_vec
        nCells = nCells - 1;
    end
    % index the number of passes
    pass = pass + 1;
end

%------------------- calculate output --------------------
% calculate change points
CPTs=cumsum(length_vec');
heights = [pop_vec'./length_vec'];

% calculate objective function value for the model
posterior = sum(glblpost(length_vec, pop_vec));

time = toc;
```

A.2 Local Search Best Divide BD1.m

```
function [time, CPTs, posterior, heights] = ...
    BD1(length_vec, pop_vec)
%-------------------------------------------------------------
%BD1b_2.m    David Barnes6/24/02
%   [time, CPTs, posterior, heights] = ...
%       BD1(length_vec, pop_vec)
%
%   A divide only local search algorithm for 1-dimensional
%   data.  The two cells with the divide factor are split
%   to create two new blocks.  Then the process repeats
%   until no divide factor greater than 0 exists.
%
%   input:  length_vec - lengths of the voronoi cells
%           pop_vec - lists the population of each cell
%
%   output: time - the time to find a solution
%           CPTs - a vector of change point times
%           heights - a vector listing the intensities
%                     of each block
%           posterior - the objective function value for
%                       the model generated by the algorithm
%-------------------------------------------------------------

tic

nCells = length(length_vec);
% the initial partition of the space:
% all cells part of a block that starts at 1, ends at nCells
begin=ones(1,nCells);
finish=nCells*begin;

for i=1:(nCells-1)
    % first calculation of divide factors
    divide(i) = divfctr1(sum(length_vec(1:i)), ...
        sum(pop_vec(1:i)), sum(length_vec(i+1:nCells)), ...
        sum(pop_vec(i+1:nCells)));
end

max_divide = 1;

%------------------ the divide algorithm ------------------
while (max_divide>0)
    % as long as a divide factor is greater than zero
    % find the largest divide factor and its index
    [max_divide index]=max(divide);

    % divide cells with the largest divide factor
```

```
    if (max_divide>0)
        frst1=begin(index);
        frst2=index+1;
        lst1=index;
        lst2=finish(index);

        begin(frst1:lst1)=frst1;
        begin(frst2:lst2)=frst2;
        finish(frst1:lst1)=lst1;
        finish(frst2:lst2)=lst2;

        % set the divide factor just done to zero
        divide(lst1) = 0;

        % compute divide factors changed by division
        for i=1:(lst1-frst1)
            divide(frst1+i-1) = ...
               divfctr1(sum(length_vec(frst1:frst1+i-1)),...
               sum(pop_vec(frst1:frst1+i-1)), ...
               sum(length_vec(frst1+i:lst1)),...
               sum(pop_vec(frst1+i:lst1)));
        end

        for i=1:(lst2-frst2)
            divide(frst2+i-1) = ...
               divfctr1(sum(length_vec(frst2:frst2+i-1)),...
               sum(pop_vec(frst2:frst2+i-1)), ...
               sum(length_vec(frst2+i:lst2)),...
               sum(pop_vec(frst2+i:lst2)));
        end
     end
end
%-------------------- calculate output --------------------
% get lengths and sizes of the blocks
INDEX = begin(nCells);
[l_block, p_block] = ...
    partition1d(length_vec, pop_vec, begin);

% get change points
CPTs=cumsum(l_block');
heights = [p_block'./l_block'];

% calculate objective function value for the model
posterior = sum(glblpost(l_block, p_block));
time = toc;
```

A.3 Local Search Best Merge/Divide BMD1.m

```
function [time, CPTs, posterior, heights] = ...
    BMD1(length_vec, pop_vec, which_start)
%-------------------------------------------------------------
%BMD1_2    David Barnes    6/22/02
%    [time, CPTs, posterior, heights] = ...
%        BMD1(length_vec, pop_vec)
%
%    A merge/divide local search algorithm for one-
%    dimensional data.  The two cells with the largest
%    merge/divide factor are acted on, then the process
%    repeats until no two cells have a m/d factor greater
%    than 0.
%
%    input:  length_vec - lengths of the voronoi cells
%            pop_vec - lists the population of each cell
%            which_start - determines the initial state:
%                0 - complete partition
%                1 - single-block partition
%                2 - random start partition
%
%    output: time - the time to find a solution
%            CPTs - a vector of change point times
%            heights - a vector listing the intensities
%                      of each block
%            posterior - the objective function value for
%                        the model generated by the algorithm
%-------------------------------------------------------------

tic
[begin, finish, lblocksum, pblocksum, ...
    mrgdivfact] = initialpartition(length_vec, ...
    pop_vec, which_start);

nCells = length(length_vec);

max_mergediv = 1;

pass = 0;

%--------------- the merge/divide algorithm ---------------

while (max_mergediv>0)
    % as long as a merge/divide factor is greater than zero

    % find the largest merge/divide factor and its index
    [max_mergediv index]=max(mrgdivfact);
```

```
    if (max_mergediv > 0)

        if begin(index) ~= begin(index+1)
            % --------- merge cells ---------
            [begin, finish, mrgdivfact, lblocksum, ...
                pblocksum] = mergeblocks1d(length_vec, ...
                pop_vec, begin, finish, mrgdivfact, ...
                index, nCells, lblocksum, pblocksum);
        else
            % --------- divide cells ---------
            [begin, finish, mrgdivfact, lblocksum, ...
                pblocksum] = divideblocks1d(length_vec, ...
                pop_vec, begin, finish, mrgdivfact, ...
                index, nCells, lblocksum, pblocksum);
        end
    end
end

%-------------------- calculate output --------------------
% get lengths and sizes of the blocks
[l_block, p_block]= partition1d(length_vec, pop_vec, begin);

% calculate change points
CPTs=cumsum(l_block');
heights = [p_block'./l_block'];

% calculate objective function value for the model
posterior = sum(glblpost( l_block, p_block));

time = toc;
```

A.4 Tabu Search tabu1d.m

```
function [time, CPTs, posterior, heights] = ...
    tabu1d(length_vec, pop_vec, which_start, tabu,cycles)
%-------------------------------------------------------------
%tabu1db_2      David Barnes    6/23/02
%   [time, CPTs, posterior, heights] = ...
%       tabu1db_2(length_vec, pop_vec, which_start, ...
%       tabu, cycles)
%
%   A merge/divide tabu search algorithm for 1-dimensional
%   data.  The action corresponding to the largest
%   merge/divide factor is performed cycles times.  A
%   running record of the best solution found so far is kept
%   and updated to be used as the final solution for the
%   algorithm.  Also, a tabu list of the last tabu#
%   operations is kept so recent actions are not undone.
%
%   input:  length_vec - lengths of the voronoi cells
%           pop_vec - lists the population of each cell
%           which_start - determines the initial state:
%               0 - complete partition
%               1 - single-block partition
%               2 - random start partition
%           tabu - the length of the tabu list
%           cycles - the stop criterion for the algorithm
%
%   output: time - the time to find a solution
%           CPTs - a vector of change point times
%           heights - a vector listing the intensities
%                     of each block
%           posterior - the objective function value for
%                       the model generated by the algorithm
%-------------------------------------------------------------
tic

nCells = length(length_vec);

[begin, finish, lblocksum, pblocksum, ...
    mrgdivfact] = initialpartition(length_vec, ...
    pop_vec, which_start);
mrgdivfact2 = mrgdivfact;

% initialize some variables and arrays
pass = 0;
invalid = zeros(1,tabu);
obj_ch1 = 0;
best = 0;
```

```
%---------------- the tabu search algorithm ----------------
while (pass <cycles)
    % find the largest merge/divide factor and index
    [max_mergediv index]=max(mrgdivfact2);

    invalid = [index invalid(1:tabu-1)];

    if begin(index)~=begin(index+1)
        % --------- merge cells ---------
        [begin, finish, mrgdivfact, lblocksum, ...
                pblocksum] = mergeblocks1d(length_vec, ...
            pop_vec, begin, finish, mrgdivfact, ...
            index, nCells, lblocksum, pblocksum);
    else
        % --------- divide cells ---------
        [begin, finish, mrgdivfact, lblocksum, ...
                pblocksum] = divideblocks1d(length_vec, ...
            pop_vec, begin, finish, mrgdivfact, ...
            index, nCells, lblocksum, pblocksum);
    end

    % assign small m/d factor to actions on the tabu list
    mrgdivfact2 = mrgdivfact;
    XX = find(invalid);
    mrgdivfact2(invalid(XX))= -inf;

    % a running total of changes to the objective function
    obj_ch1 = max_mergediv + obj_ch1;

    % keep a record of best solution so far
    if obj_ch1 > best
        % update best solution
        best = obj_ch1;
        % retain some return values
        best_begin = begin;
        best_pass = pass;
    end
    pass=pass+1;    %index the number of passes
end
%-----------------------------------------------------------
%------------------- calculate output ---------------------
% get lengths and sizes of the blocks
[l_block, p_block] = ...
    partition1d(length_vec, pop_vec, best_begin);
% calculate change points
CPTs=cumsum(l_block');
heights = [p_block'./l_block'];
% calculate objective function value for the model
posterior = sum(glblpost(l_block, p_block));
time = toc;
```

A.5 Simulated Annealing SA1d.m

```
function [time, CPTs, posterior, heights] =...
    SA1d(length_vec, pop_vec, which_start, ...
    T, cycles, LK, dec, incr)
%-------------------------------------------------------------
%SA1db_2.m      David Barnes    6/23/02
%   [time, CPTs, posterior, heights] =...
%       SA1db_2(length_vec, pop_vec, which_start, ...
%       T, cycles, LK, dec, incr)
%
%   A merge/divide simulated annealing algorithm for
%   1-dimensional data.  An action is chosen at random,
%   performed always if it benefits the objective
%   function and with a probability if it does not benefit
%   the objective function.
%
%   input:  length_vec - lengths of the voronoi cells
%           pop_vec - lists the population of each cell
%           which_start - determines the initial state:
%               0 - complete partition
%               1 - single-block partition
%               2 - random start partition
%           T - the initial value for the control parameter
%           cycles - the stop criterion for the algorithm
%           LK - the number of passes of the inner algorithm
%           dec - the decrement constant
%           incr - the increment used to generate the
%                  length of each subsequent chain LK(new)
%
%   output: time - the time to find a solution
%           CPTs - a vector of change point times
%           heights - a vector listing the intensities
%                     of each block
%           posterior - the objective function value for
%                       the model generated by the algorithm
%-------------------------------------------------------------
tic

[begin, finish, lblocksum, pblocksum, ...
    mrgdivfact] = initialpartition(length_vec, ...
    pop_vec, which_start);
nCells = length(length_vec);

accept = 0;
pass = 0;
```

```
%------------ the simulated annealing algorithm ------------
while (pass<cycles)
    for iChain = 1:LK
        index=ceil((nCells - 1)*rand);
        % consider acting on the two cells picked at random
        value = mrgdivfact(index);

        if (value>0) accept = 1;
        else
            test = exp(value/T);
            q = rand;
            if (q<test) accept = 1;
            end
        end

        if accept == 1
            if begin(index)~=begin(index+1)
                % --------- merge cells ---------
                [begin, finish, mrgdivfact, lblocksum, ...
                    pblocksum] = ...
                    mergeblocks1d(length_vec, pop_vec, ...
                    begin, finish, mrgdivfact, index, ...
                    nCells, lblocksum, pblocksum);
            else
                % --------- divide cells ---------
                [begin, finish, mrgdivfact, lblocksum, ...
                    pblocksum] = ...
                    divideblocks1d(length_vec, pop_vec, ...
                    begin, finish, mrgdivfact, index, ...
                    nCells, lblocksum, pblocksum);
            end
        end
        accept = 0;
    end

    % update algorithm parameters
    T=dec*T;
    LK=LK + incr;
    pass = pass+1;
end
%-------------------------------------------------------------
%-------------------- calculate output ---------------------
% get lengths and sizes of the blocks
[l_block, p_block]= partition1d(length_vec, pop_vec, begin);
% calculate change points
CPTs=cumsum(l_block');
heights = [p_block'./l_block'];
% calculate objective function value for the model
posterior = sum(glblpost( l_block, p_block));
time = toc;
```

A.6 Dynamic Programming Algorithm dynamic1d.m

```
function [time, CPTs, posterior, heights] = ...
    dynamic1d(length_vec, pop_vec)
%-------------------------------------------------------------
%dynamic1d      David Barnes    6/23/02
%
%   An algorithm based on dynamic programming that finds
%   the optimal partition of the data.
%
%   input:  length_vec - lengths of the voronoi cells
%           pop_vec - lists the population of each cell
%
%   output: time - the time to find a solution
%           CPTs - a vector of change point times
%           heights - a vector listing the intensities
%                     of each block
%           posterior - the objective function value for
%                       the model generated by the algorithm
%-------------------------------------------------------------
tic
nCells=length(pop_vec);

% at step i the best partition of the first i cells is found

% best(i) = function value for the opt. part. of 1st i cells
best = glblpost(length_vec(1),pop_vec(1));

% last start(i) is the index of the first cell in the last
% block of the optimal partition of the first i cells
last_start = 1;

for R=2:nCells
    % compute the function value for merging
    % 1..R, 2..R, ... R-1..R, R in merged
    merged = [fliplr(glblpost(cumsum(length_vec(R:-1:1)),...
            cumsum(pop_vec(R:-1:1))))];
    [best(R), last_start(R)] = max([0 best]+merged);
end
%------------------- calculate output --------------------
% get lengths and sizes of the blocks
[l_block,p_block] = ...
   partition1d(length_vec, pop_vec, last_start);
newL = length(l_block);
% calculate objective function value for the model
posterior = sum(glblpost(l_block(1:newL), p_block(1:newL)));
% get change points
CPTs = cumsum(l_block');
heights = [p_block'./l_block'];
time = toc;
```

A.7 Support Programs I

condition1d.m

```
function [length_vec, pop_vec] = condition1d(data)
%-------------------------------------------------------------
%condition1d    David Barnes    6/15/02
%   [length_vec, pop_vec] = condition1d(data)
%
%   Generates the inputs for the algorithms for the one-
%   dimensional problems.  This function is able to
%   accommodate multiple points, or equal entries in data.
%
%   input:  data - a non-decreasing list of numbers
%
%   output: length_vec - lengths of the voronoi cells
%           pop_vec - populations of the voronoi cells
%-------------------------------------------------------------
nCells = length(data);
% generate first and last data points
firstval = (3*data(1)-data(2))/2;
endval = (3*data(nCells)-data(nCells-1))/2;
% data augmented so firstval & endval are the endpoints
newData = [firstval data endval];
length_vec(1) = (newData(3) - newData(1))/2;
pop_vec(1)=1;
for i=2:nCells
    if data(i)~=data(i-1)
        length_vec(i) = (newData(i+2) - newData(i))/2;
        pop_vec(i) = 1;
        last = i;
    else
        length_vec(last) = length_vec(last) + ...
            (newData(i+2) - newData(i))/2;
        pop_vec(last) = pop_vec(last)+1;
    end
end
newLength = length(pop_vec);
tempsz = pop_vec(1);
templg = length_vec(1);

% eliminate null cells
for i = 2:newLength
    if pop_vec(i)~=0
        tempsz = [tempsz pop_vec(i)];
        templg = [templg length_vec(i)];
    end
end
pop_vec = tempsz;
length_vec = templg;
```

initialpartition.m

```
function [begin, finish, lblocksum, pblocksum, ...
    mrgdivfact] = initialpartition(length_vec, ...
    pop_vec, which_start)
%-------------------------------------------------------------
%initialpartition    David Barnes    9/19/02
%    [begin, finish, lblocksum, pblocksum, ...
%        mrgdivfact] = initialpartition(length_vec, ...
%        pop_vec, which_start)
%
%    A program to initialize the arrays necessary for the
%    one-dimensional merge divide algorithms.
%
%    input:  length_vec - lengths of the initial cells
%            pop_vec - lists the population of each cell
%            which_start - determines the initial state:
%                0 - complete partition
%                1 - single-block partition
%                2 - random start partition
%
%    output: begin - indices of the first cell in the block
%                    to which every cell belongs
%            finish - indices of the last cell in the block
%                     to which every cell belongs
%            lblocksum - lblocksum(i) = length of the block
%                        to which cell i belongs
%            pblocksum - pblocksum(i) = population of the
%                        block to which cell i belongs
%            mrgdivfact - mrgdivfact(i) = merge or divide
%                         factor determined by the adjacency
%                         i, i+1
%-------------------------------------------------------------
k = length(length_vec);
if which_start==0
    % start with complete partition
    % each cell constitutes its own block
    begin = linspace(1,k,k);
    finish = begin;
    % first calculate merge/divide factors for all
    % adjacent cells (all merge factors)
    mrgdivfact = mrgfctr1(length_vec(1:k-1), ...
        pop_vec(1:k-1), length_vec(2:k), pop_vec(2:k));
    lblocksum = length_vec;
    pblocksum = pop_vec;
end
%-------------------------------------------------------------
if which_start==1
    % start will single block partition
    % all cells part of a block that starts at 1, ends at k
```

```
    begin = ones(1,k);
    finish = k*begin;
    for i=1:(k-1)
    % first calculation of merge/divide factors all
    % adjacent cells (all divide factors)
        mrgdivfact(i) = -mrgfctr1(sum(length_vec(1:i)), ...
            sum(pop_vec(1:i)), sum(length_vec(i+1:k)), ...
            sum(pop_vec(i+1:k)));
    end
    lblocksum(1:k) = sum(length_vec);
    pblocksum(1:k) = sum(pop_vec);
end
%-------------------------------------------------------------
if which_start==2
    % start with a random partition of the data
    % generate random blocks for the start of the algorithm
    first=1;
    last=1;
    while last<k
        % find a random number at which to end a block
        rnmbr=round((k-first)*rand)+first;
        % start the block at first and end at number found
        begin(first:rnmbr)=first;
        finish(first:rnmbr)=rnmbr;
        first=rnmbr+1;  % new first = last end + 1
        last=rnmbr;     % used in stopping the while loop
    end
    % first calculation of merge/divide factors for all
    % adjacent cells
    for i=1:k-1
        if begin(i)~=begin(i+1) % cells in different blocks
            mrgdivfact(i) = mrgfctr1(length_vec(i), ...
                pop_vec(i), length_vec(i+1), pop_vec(i+1));
            lblocksum(begin(i):finish(i)) = ...
                sum(length_vec(begin(i):finish(i)));
            lblocksum(begin(i+1):finish(i+1)) = ...
                sum(pop_vec(begin(i+1):finish(i+1)));
            pblocksum(begin(i):finish(i)) = ...
                sum(length_vec(begin(i):finish(i)));
            pblocksum(begin(i+1):finish(i+1)) = ...
                sum(length_vec(begin(i+1):finish(i+1)));
        else
            mrgdivfact(i) = ...
                -mrgfctr1(sum(length_vec(begin(i):i)),...
                sum(pop_vec(begin(i):i)), ...
                sum(length_vec(i+1:finish(i))),...
                sum(pop_vec(i+1:finish(i))));
        end
    end
end
```

mergeblocks1d.m

```
function [begin, finish, mrgdivfact, lblocksum, ...
    pblocksum] = mergeblocks1d(length_vec, pop_vec, ...
    begin, finish, mrgdivfact, index, k, ...
    lblocksum, pblocksum)
%-------------------------------------------------------------
%mergeblocks1d      David Barnes    9/24/02
%   [begin, finish, mrgdivfact, lblocksum, ...
%       pblocksum] = mergeblocks1d(length_vec, pop_vec, ...
%       begin, finish, mrgdivfact, index, k, ...
%       lblocksum, pblocksum)
%
%   A function to merge blocks for merge/divide algorithms
%   processing one-dimensional data.
%
%   input:  length_vec - lengths of the voronoi cells
%           pop_vec - lists the population of each cell
%           begin - indices of the first cell in the block
%                   to which every cell belongs
%           finish - indices of the last cell in the block
%                    to which every cell belongs
%           mrgdivfact - mrgdivfact(i) = merge or divide
%                        factor determined by the adjacency
%                        i, i+1
%           index - the index of the merge: blocks index,
%                   index+1
%           k - size of the original data
%           lblocksum - lblocksum(i) = length of the block
%                       to which cell i belongs
%           pblocksum - pblocksum(i) = population of the
%                       block to which cell i belongs
%
%   output: updated versions of the input arrays:
%           begin, finish, mrgdivfact, lblocksum, pblocksum
%-------------------------------------------------------------

% find the first and last cells of the block
frst = begin(index);
lst  = finish(index+1);

begin(frst:lst) = frst;
finish(frst:lst)= lst;

% compute length and pop of new block
lblocksum(frst:lst) = sum(length_vec(frst:lst));
pblocksum(frst:lst) = sum(pop_vec(frst:lst));
```

```
if lst<k
    mrgdivfact(lst)=mrgfctr1(lblocksum(frst),...
        pblocksum(frst), lblocksum(lst+1), ...
        pblocksum(lst+1));
end

if frst>1
    mrgdivfact(begin(index)-1)= ...
        mrgfctr1(lblocksum(frst-1), ...
        pblocksum(frst-1),lblocksum(frst), ...
        pblocksum(frst));
end

% compute divide factors for cells in the block
for i = 1:(lst-frst)
    lhs_length = sum(length_vec(frst:frst+i-1));
    lhs_pop = sum(pop_vec(frst:frst+i-1));
    rhs_length = sum(length_vec(frst+i:lst));
    rhs_pop = sum(pop_vec(frst+i:lst));
    mrgdivfact(frst+i-1) = ...
        -mrgfctr1(lhs_length, lhs_pop, ...
        rhs_length, rhs_pop);
end
```

divideblocks1d.m

```
function [begin, finish, mrgdivfact, lblocksum, ...
    pblocksum] = divideblocks1d(length_vec, pop_vec, ...
    begin, finish, mrgdivfact, index, k, lblocksum, ...
    pblocksum)
%-------------------------------------------------------------
%divideblocks1d     David Barnes    9/24/02
%    [begin, finish, mrgdivfact, lblocksum, ...
%        pblocksum] = divideblocks1d(length_vec, pop_vec, ...
%        begin, finish, mrgdivfact, index, k, lblocksum, ...
%        pblocksum)
%
%    A function to divide blocks for merge/divide algorithms
%    processing one-dimensional data.
%
%    input:  length_vec - lengths of the voronoi cells
%            pop_vec - lists the population of each cell
%            begin - indices of the first cell in the block
%                    to which every cell belongs
%            finish - indices of the last cell in the block
%                     to which every cell belongs
%            mrgdivfact - mrgdivfact(i) = merge or divide
%                         factor determined by the adjacency
%                         i, i+1
%            index - the index of the divide: divide block
%                    at index, index+1
%            k - size of the original data
%            lblocksum - lblocksum(i) = length of the block
%                        to which cell i belongs
%            pblocksum - pblocksum(i) = population of the
%                        block to which cell i belongs
%
%    output: updated versions of the input arrays:
%            begin, finish, mrgdivfact, lblocksum, pblocksum
%-------------------------------------------------------------

% find the first and last cells in each of the new blocks
frst1=begin(index);
frst2=index+1;
lst1=index;
lst2=finish(index);

begin(frst1:lst1)=frst1;
begin(frst2:lst2)=frst2;
finish(frst1:lst1)=lst1;
finish(frst2:lst2)=lst2;
```

```
lblocksum(frst1:lst1) = ...
    sum(length_vec(frst1:lst1));
lblocksum(frst2:lst2) = ...
    sum(length_vec(frst2:lst2));
pblocksum(frst1:lst1) = ...
    sum(pop_vec(frst1:lst1));
pblocksum(frst2:lst2) = ...
    sum(pop_vec(frst2:lst2));

% compute merge factors changed by divide
if lst2<k
    mrgdivfact(lst2) = ...
        mrgfctr1(lblocksum(lst2), ...
        pblocksum(lst2),...
        lblocksum(lst2+1), pblocksum(lst2+1));
end

if frst1>1
    mrgdivfact(frst1-1) = ...
        mrgfctr1(lblocksum(frst1-1),...
        pblocksum(frst1-1), ...
        lblocksum(frst1), pblocksum(frst1));
end

mrgdivfact(lst1) = mrgfctr1(lblocksum(lst1), ...
    pblocksum(lst1), lblocksum(frst2), ...
    pblocksum(frst2));

% compute divide factors changed by divide
for i=1:(lst1-frst1)
    mrgdivfact(frst1+i-1) = ...
        -mrgfctr1(sum(length_vec ...
        (frst1:frst1+i-1)),...
        sum(pop_vec(frst1:frst1+i-1)), ...
        sum(length_vec(frst1+i:lst1)),...
        sum(pop_vec(frst1+i:lst1)));
end

for i=1:(lst2-frst2)
    mrgdivfact(frst2+i-1) = ...
        -mrgfctr1(sum(length_vec...
        (frst2:frst2+i-1)),...
        sum(pop_vec(frst2:frst2+i-1)), ...
        sum(length_vec(frst2+i:lst2)),...
        sum(pop_vec(frst2+i:lst2)));
end
```

mrgfctr1.m

```
function m = mrgfctr1(length1, size1, length2, size2)
%-------------------------------------------------------------
%mrgfctr1.m    David Barnes 4/02/02
%   m = mrgfctr1(length1, size1, length2, size2)
%
%   Computes the effect merging two blocks has on the log
%   of the objective function
%
%   Input:  length1 - volume of the first block
%           size1 - size of the first block
%           length2 - volume of the second block
%           size2 - size of the second block
%
%   Output: m - the merge factor for the two blocks
%-------------------------------------------------------------
VAL_IF_MERGE = glblpost(length1 + length2, size1 + size2);
P1 = glblpost(length1, size1);
P2 = glblpost(length2, size2);
m = VAL_IF_MERGE - P1 - P2;
```

glblpost.m

```
function value = glblpost(V, S)
%-------------------------------------------------------------
%glblpost     David Barnes    7/02/02
%  value = glblpost(V, S)
%
%  Objective function. In the case a block has a size
%  greater than its volume an objective function value of
%  zero is returned.
%
%  input:  V - the area of a block
%          S - the size of a block
%  output: value - objective function value of the block
%-------------------------------------------------------------
x = V-S+1;
loc = find(x<0);
if loc V(loc) = 100000;
end
if V < S-1 value = 0;
else
    value = GAMMALN(S+1) + GAMMALN(V-S+1) - GAMMALN(V+2);
end
if loc value(loc) = 0;
end
```

partition1d.m

```
function [block_lengths, block_sizes] = ...
    partition1d(length_vec, pop_vec, last_start)
%-------------------------------------------------------------
%partition1d    David Barnes 6/20/02
%
%   Partitions one-dimensional data to blocks as prescribed
%   by an algorithm.
%
%   input:  length_vec - original lengths of cells
%           pop_vec - original populations of cells
%           last_start - indices to the first cell in the
%                        block to which cell(i) belongs
%
%   output: block_lengths - lengths of the new blocks
%           block_sizes - sizes of the new blocks
%-------------------------------------------------------------

lastCell = length(length_vec);
index = last_start(lastCell);

while (index)
    % sum the length_vecs and pop_vecs for the last merge
    block_lengths(lastCell) =...
        sum(length_vec(index:lastCell));
    block_sizes(lastCell) = sum(pop_vec(index:lastCell));

    % delete extraneous cells
    block_lengths(index:lastCell-1) = [];
    block_sizes(index:lastCell-1) = [];

    % the new last index (for the next merge)
    lastCell = index-1;
    if lastCell>0 index = last_start(lastCell);
        % if more merges to do, find the index of the first
        % cell in the previous merge
    else
        % otherwise end while loop
        index = 0;
     end
end
```

plotblocks.m

```
function [X,Y] = plotblocks(changePoints, heights)
%-------------------------------------------------------------
%plotblocks     David Barnes    9/24/02
%   [X, Y] = plotblocks(changePoints, heights)
%
%   A function to plot the block structure of a data set
%   based on the return values from an algorithm.  Plots
%   the intensity, or height, for blocks.  The x and y
%   values used in the plot are also returned so they can
%   be used in another application if desired.
%
%   input:  changePoints - the change points returned by an
%                          algorithm for the data set
%           heights - the intensity of the blocks
%                     (population/length)
%
%   output: X - the x-values used in plotting the blocks
%           Y - the y-values used in plotting the blocks
%-------------------------------------------------------------

nCells = length(changePoints);

X(1) = 0;

for i = 1:nCells-1
    X(2*i:2*i+1) = [changePoints(i), changePoints(i)];
    Y(2*i-1:2*i) = [heights(i), heights(i)];
end

X = [X changePoints(nCells)];
Y = [Y heights(nCells) heights(nCells)];

plot(X,Y,'k-')
xlabel('TIME')
ylabel('Intensity (size(i)/area(i))')
axis([0 changePoints(nCells) 0 1]);
```

Appendix B Algorithms for Two-Dimensional Data

This section contains all the function programs for the analysis of two-dimensional data. Just as in appendix A, the final section contains all the programs necessary to support these algorithms, and one program, colour2d5.m, to visualize results. To find the rank of a sparse matrix the built-in rank function was adapted by changing the line s = svd(A) to s = svds(A). This new function called rank2 is used by the merge/divide algorithms but is not included here.

B.1 Local Search Best Merge BM2.m

```
function [time, newa_vec, news_vec, contains2, ...
    posterior] = BM2b_2(area_vec1, size_vec1, adjsp1, C1)
%-------------------------------------------------------------
%BM2b_2     David Barnes    8/01/02
%    [time, newa_vec, news_vec, contains2, posterior]=...
%        BM2b_2(area_vec1, size_vec1, adjsp1, C1)
%
%    A best merge local search (greedy merge-only)
%    implementation.  As the algorithm progresses, all
%    information about the block structure of the space
%    is updated.  The block with the larger index is added to
%    the one with the smaller index, and then is deleted.
%
%    input:  adjsp -- a sparse adjacency matrix for the data
%            area_vec -- areas of all the Voronoi cells
%            size_vec -- sizes of the cells
%            C1 -- a prior on the number of blocks in the
%                  model
%    output: time -- the time to find the solution
%            newa_vec -- the areas of the output blocks
%            news_vec -- the sizes of the output blocks
%            contains2 -- lists the cells in each block
%            posterior -- the posterior for the model
%-------------------------------------------------------------
tic
global area_vec size_vec blocknum
global adjsp merge_mat
global C
area_vec = area_vec1;
size_vec = size_vec1;
adjsp = adjsp1;
```

```
C = C1;

% preserves the original values in the arrays
temparea=area_vec;
tempsize=size_vec;
tempadj=adjsp;

% compute merge factors for all adjacent cells
nBlocks=length(temparea);
nBlocks1 = nBlocks;
merge_mat = sparse(zeros(nBlocks));

for i=1:nBlocks
    newMrg = find(tempadj(i,:));
    newL = length(newMrg);
    merge_mat(i,newMrg) = ...
        mrgfctr2(temparea(i)*ones(1,newL), ...
        tempsize(i)*ones(1,newL), temparea(newMrg),...
        tempsize(newMrg),C);
end

for i=1:nBlocks
    % each cell constitutes its own block
    contains{i}=i;
end

% max is taken twice: first it returns a row vector
% of max values for each col; the second time it returns
% the index of the max of the new row and that max value
[best,col]=max(max(merge_mat));
[best,row]=max(merge_mat(:,col));
pass = 0;
%---------------------- The Algorithm ----------------------
while best>0
    % as long as as the largest merge factor is
    % greater than 0, keep merge/dividing (greedy alg.)
    pass = pass + 1;
    least=min(row,col);
    maximum=max(row,col);

    % merge blocks to the one with smallest index
    temparea(least) = temparea(least) + temparea(maximum);
    tempsize(least) = tempsize(least) + tempsize(maximum);

    % Update the list of cells contained in the new block
    contains{least} = [contains{least} contains{maximum}];

    % Delete the unused block
    contains{maximum} = [0];
    temparea(maximum) = 0;
    tempsize(maximum) = 0;
```

```
    %Reset adjacencies
    newA = find(tempadj(maximum,:));
    tempadj(least,[newA]) = 1;
    tempadj(:,least) = tempadj(least,:)';
    tempadj(least,least) = 0;

    % adjacency matrix reset to reflect deleting cell
    tempadj(maximum,:) = 0;
    tempadj(:,maximum) = 0;

    % invalid merge factors are deleted
    merge_mat(maximum,:) = 0;
    merge_mat(:,maximum) = 0;
    nBlocks = length(find(temparea));
    if nBlocks>1
        % if all cells do not belong to the same block
        % merge factors for the new merge block are computed
        newMrg = find(tempadj(least,:));
        newL = length(newMrg);
        merge_mat(least,newMrg) = ...
            mrgfctr2(temparea(least)*ones(1,newL),...
            tempsize(least)*ones(1,newL), ...
            temparea(newMrg), tempsize(newMrg), C);

        merge_mat(:,least) = merge_mat(least,:)';

        % same procedure to find the largest merge factor
        [best,col]=max(max(merge_mat));
        [best,row]=max(merge_mat(:,col));
    else
        % quit loop
        best = -1;
    end
end

blocksAt = find(temparea);
j=1
for i = 1:length(blocksAt)
    contains2{i} = contains{blocksAt(i)};
    newa_vec(i) = temparea(blocksAt(i));
    news_vec(i) = tempsize(blocksAt(i));
end
L = length(news_vec);
% calculate the objective function value for the model
posterior=sum(glblpost2(newa_vec(1:L),news_vec(1:L),C));
time = toc;
```

B.2 Local Search Best Merge/Divide BMD2.m

```
function [time, newa_vec, news_vec, contains2,...
    posterior] = BMD2b_2(area_vec1, size_vec1, ...
    adjsp1, pref, C1)
%-------------------------------------------------------------
% BMD2b_2   David Barnes    9/17/02
%   [time, newa_vec, news_vec, contains2, posterior] ...
%       = BMD2b_2(pref, C1)
%
%   A greedy merge divide implementation that works on any
%   given data given the volumes (sizes 2-dim) and
%   sizes of the voronoi cells, and the adjacency matrix
%   describing the relationship between the cells. At each
%   step the largest merge/divide factor is found, and
%   carried out provided it benefits the objective
%   function and a division is not the removal of a cell
%   that corresponds to a cut vertex.  The 'cut' problem
%   is addressed in the merge/divide function.
%
%   input:  adjsp -- a sparse adjacency matrix for the data
%           area_vec -- areas of all the Voronoi cells
%           size_vec -- sizes of the cells
%           pref -- 1 for a bottom-up start
%                   2 for a top-down start
%           C1 -- a prior on the number of blocks in the
%                 model
%   output: time -- the time to find the solution
%           newa_vec -- the areas of the output blocks
%           news_vec -- the sizes of the output blocks
%           contains2 -- lists the cells in each block
%           posterior -- the posterior for the model
%-------------------------------------------------------------
tic

global area_vec size_vec blocknum
global adjsp merge_mat
global C

NPOINTS = length(area_vec);

area_vec = area_vec1;
size_vec = size_vec1;
adjsp = adjsp1;
C = C1;

nPass = 0;
% row & col are indices to the block(s) to be acted on
row = 0;
col = 0;
```

```
% best is the largest merge/divide factor
best = 1;
%---------------------- The Algorithm ----------------------
while (best>0)
    % as long as as the largest merge/divide factor is
    % greater than 0, keep merge/dividing (greedy alg.)
    nPass = nPass + 1;

    opNumber = nPass;
    [newContents, lables] =...
        mergedivideblocks4(opNumber, row, col, pref);
    nAdjustments = length(lables);

    for iAdjust = 1:nAdjustments
        % update those blocks changed by the last m/d
        contains{lables(iAdjust)} =...
            newContents{iAdjust};
    end

    % max is taken twice: first it returns a row vector
    % of max values for each col; the second time it returns
    % the index of the max of the new row and that max value
    [best,col]=max(max(merge_mat));
    [best,row]=max(merge_mat(:,col));

end
%-------------------------------------------------------------

blocksAt = unique(blocknum);

for i = 1:length(blocksAt)
    contents = find(blocknum == blocksAt(i));
    contains2{i} = contents;
    newa_vec(i) = sum(area_vec(contents));
    news_vec(i) = length(contents);
end

L=length(news_vec);
% calculate the objective function value for the model
posterior=sum(glblpost2(newa_vec(1:L),news_vec(1:L),C));
time = toc;
```

B.3 Tabu Search tabu2d.m

```
function [time, newa_vec, news_vec,contains2, posterior]=...
    tabu2d_3(area_vec1, size_vec1, adjsp1, pref, ...
    tabu, cycles, C1)
%-------------------------------------------------------------
%tabu2dv1_2     David Barnes    9/18/02
%[time, newa_vec, newsvec, contains2, posterior]=...
%   tabu2d_3(pref, tabu, cycles)
%
%   A tabu search implementation that works for
%   any type of data given the volumes (areas 2-dim) and
%   sizes of the voronoi cells, and the adjacency matrix
%   describing the relationship between the cells.  A
%   record is kept of the last tabu# actions performed
%   that can be undone.  If an action cannot be undone a
%   null entry is entered in the tabu list so the other
%   entries in the list will remain there for the correct
%   number of passes.  The number of values to enter on
%   the list varies, since a divide corresponds to a single
%   entry in the merge matrix, while a merge corresponds to
%   two entries in the list.  So a record of how many
%   entries should be on the list, and should be taken off
%   at each step, is maintained.
%
%   input:  adjsp -- a sparse adjacency matrix for the data
%           area_vec -- areas of all the Voronoi cells
%           size_vec -- sizes of the cells
%           pref -- 1 for a bottom-up start
%                   2 for a top-down start
%           tabu - the length of the tabu list
%           cycles - the stop criterion for the algorithm
%           C1 -- a prior on the number of blocks in the
%                 model
%   output: time -- the time to find the solution
%           newa_vec -- the areas of the output blocks
%           news_vec -- the sizes of the output blocks
%           contains2 -- lists the cells in each block
%           posterior -- the posterior for the model
%-------------------------------------------------------------

tic % start a clock

global area_vec size_vec blocknum
global adjsp merge_mat
global C

area_vec = area_vec1;
size_vec = size_vec1;
adjsp = adjsp1;
```

```
C = C1;

NPOINTS = length(area_vec);

% set a default location to be inserted in the tabu list
% when the action performed cannot be undone
default = find(adjsp(1,2:NPOINTS)==0)+1;
default2 = min(default);

% initialize the tabu list
invalid = [];
add_to_list = [];

% initialize variables to keep max m/d factor and a record
% of the changes made to the objective function
max_mergediv = 0;
obj_ch1 = 0;

% best keeps a record of the best value for the objective
% function so far
best = -1;
nPass = 0;

row = 0;
col = 0;

Incr = zeros(1,tabu);
%---------------------- The Algorithm ----------------------
while (nPass < cycles)
    % continue until the stop criterion is reached

    nPass = nPass +1;
    % ---- merge/divide blocks ----
    opNumber = nPass;
    [newContents, lables] =...
        mergedivideblocks4(opNumber,row,col,pref);
    nAdjustments = length(lables);

    for iAdjust = 1:nAdjustments
        contains{lables(iAdjust)} = newContents{iAdjust};
    end
    % ----------------------------

    % ------ update tabu list -----
    t_incr = sum(Incr);
    invalid = [add_to_list; invalid];
    bb = length(invalid);

    if (length(invalid) - t_incr) > tabu
        invalid((bb - Incr(tabu)):bb,:) = [];
    end
```

```
    merge_mat2 = merge_mat;

    for i = 1: length(invalid)
        x1 = invalid(i,1);
        x2 = invalid(i,2);
        merge_mat2(x1,x2) = -inf;
    end

    add_to_list = [];
    % ----------------------------

    % keep a running total of changes to the
    % objective function
    obj_ch1 = max_mergediv + obj_ch1;

    % keep a record of 'best' solution so far
    if obj_ch1 > best
        % update best solution
        best = obj_ch1;
        % update best partition found
        best_contains = contains;
    end

    % Find the adjacency that has the largest m/d factor
    [I J V] = find(merge_mat2);
    [max_mergediv loc]= max(V);

    row = I(loc);
    col = J(loc);
end
%-------------------------------------------------------------

%------------------- set return values --------------------
blocksAt = unique(blocknum);

for i = 1:length(blocksAt)
    contents = find(blocknum == blocksAt(i));
    contains2{i} = contents;
    newa_vec(i) = sum(area_vec(contents));
    news_vec(i) = length(contents);
end

L=length(news_vec);
% calculate the objective function value for the model
posterior=sum(glblpost2(newa_vec(1:L),news_vec(1:L),C));
time = toc;
```

B.4 Simulated Annealing SA2d.m

```
function [time, newa_vec, news_vec,contains2, posterior]=...
    SA2db_3(area_vec1, size_vec1, adjsp1, pref, T, ...
    cycles, LK, dec, incr, C1)
%-------------------------------------------------------------
% SAdb_2  David Barnes 9/12/02
%   [time, newa_vec, news_vec,contains2, posterior] = ...
%       SA2db_3(pref, T, cycles, LK, dec)
%
%   A simulated annealing implementation that works for
%   any type of data given the volumes (areas 2-dim) and
%   sizes of the voronoi cells, and the adjacency matrix
%   describing the relationship between the cells.
%   Locates actions to propose by searching the non-zero
%   entries of the matrix of merge/divide factors.
%
%   input:  adjsp -- a sparse adjacency matrix for the data
%           area_vec -- areas of all the Voronoi cells
%           size_vec -- sizes of the cells
%           pref -- 1 for a bottom-up start
%                   2 for a top-down start
%           T - the initial value for the control parameter
%           cycles - the stop criterion for the algorithm
%           LK - the number of passes of the inner algorithm
%           dec - the decrement constant
%           incr - the increment used to generate the
%                  length of each subsequent chain LK(new)
%           C1 - a prior on the number of blocks in the
%                model
%   output: time -- the time to find the solution
%           newa_vec -- the areas of the output blocks
%           news_vec -- the sizes of the output blocks
%           contains2 -- lists the cells in each block
%           posterior -- the posterior for the model
%-------------------------------------------------------------
tic
global area_vec size_vec blocknum
global adjsp merge_mat
global C
area_vec = area_vec1;
size_vec = size_vec1;
adjsp = adjsp1;
C = C1;
NPOINTS = length(area_vec);
% initialize variables
accept = 1;
row = 0;
col = 0;
nPass = 1;
```

```
%---------------------- The Algorithm ----------------------
while ( nPass < cycles+1)
    for j = 1:LK+1
        % ---- merge/divide blocks ----
        if (accept ==1)
            opNumber = nPass*j;
            [newContents, lables] =...
                mergedivideblocks4(opNumber,row,col,pref);
            nAdjustments = length(lables);
            for iAdjust = 1:nAdjustments
                contains{lables(iAdjust)} =...
                    newContents{iAdjust};
            end
        end
        % ----------------------------
        %-- locate potential actions --
        [I,J]=find(merge_mat);
        choices = length(I);
        % pick one at random
        index=ceil(choices*rand);
        row = I(index);
        col = J(index);
        %---- the decision process ----
        if merge_mat(row,col)>0 accept=1;
        else
            test_val = exp(merge_mat(row,col)/T);
            q = rand;
            if q<test_val accept=1;
            else accept=0;
            end
        end
        %-----------------------------
    end
    % ------- update parameters -------
    LK = LK + incr;
    T = dec*T;
    nPass = nPass + 1;
end
%-------------------------------------------------------------
%------------------- set return values ---------------------
blocksAt = unique(blocknum);
for i = 1:length(blocksAt)
    contents = find(blocknum == blocksAt(i));
    contains2{i} = contents;
    newa_vec(i) = sum(area_vec(contents));
    news_vec(i) = length(contents);
end
L=length(news_vec);
% calculate the objective function value for the model
posterior = sum(glblpost2(newa_vec(1:L),news_vec(1:L),C));
time = toc;
```

B.5 Support Programs II

condition2d.m

```
function [area_vec,size_vec,adj] = ...
    condition2d(data, boundaries, borders)
%-------------------------------------------------------------
%condition2d    David Barnes    8/03/02
%   [area_vec, size_vec, adjsp] = ...
%       condition2d(data, boundaries, borders)
%
%   Generates area, size, and adjacency arrays for
%   2-dimensional data based on the Voronoi tessellation
%   of the space and the delaunay triangulation of the
%   data.  Boundaries and borders define the rectangular
%   bounding region imposed on the tessellation.
%   The neighbors of cells are determined by the delaunay
%   triangulation.  Some of the boundary cells may appear
%   to be disconnected when viewed in the bounded region,
%   but are neighbors in the adjacency matrix since they
%   meet somewhere outside that area.  The built-in Matlab
%   function polyarea is used to generate the areas of
%   the regions.  This does not accommodate multiple
%   entries in data being equal.
%
% input:  data -- an nx2 matrix ith row is (x(i),y(i)),
%                 an ordered pair for a 2-d data point
%         boundaries -- the minimum and maximum allowable
%                       x & y values for any polygon
%                       vertices.
%                       Give as [minx, maxx, miny, maxy]
%         borders -- four vertices well outside the bounded
%                    region added so no data point will be
%                    contained in an unbounded voronoi cell
%
% output: area_vec -- areas of the voronoi cells
%         size_vec -- sizes of the voronoi cells (1 for all)
%         adjsp -- an adjacency matrix determined by the
%                  graph generated by the delaunay
%                  triangulation of the data
%-------------------------------------------------------------

bb = length(data);
%-------------------------------------------------------------
% generate an adjacency matrix
% tri = a trilength x 3 matrix of indices to the rows of
% data that give vertices to triangles from the delaunay
% triangulation
```

```
adj = sparse(zeros(bb));
tri = delaunay(data(:,1),data(:,2));

for i=1:length(tri)
    % check each row and record neighboring cells
    adj(tri(i,1),tri(i,2)) = 1;
    adj(tri(i,1),tri(i,3)) = 1;
    adj(tri(i,2),tri(i,3)) = 1;
    adj(tri(i,2),tri(i,1)) = 1;
    adj(tri(i,3),tri(i,1)) = 1;
    adj(tri(i,3),tri(i,2)) = 1;
end

%-------------------------------------------------------------
% find the areas of each Voronoi cell
min_x = boundaries(1); max_x = boundaries(2);
min_y = boundaries(3); max_y = boundaries(4);

[v,c]=voronoin([data; borders]);
% v = an nx2 matrix of vertices of the voronoi diagram
% for the data
% c{i} = a row vectors that is the indices of the vertices
% for the ith voronoi cell.
% To access the ith entry in c, use c{i}.
% For the jth entry of c{i}, use c{i}(j).

%generate area and size vectors
for i=1:bb
    % indices used to generate a matrix of actual vertices
    xx = v(c{i},1);
    yy = v(c{i},2);
    if min_x <=xx & xx<=max_x & min_y<=yy & yy<=max_y
        % vertices all within the bounded region
        vertices = v(c{i},:);
    else
        % vertex is not within the bounded region, use
        % function to find intersection between bounded
        % region and the polygon
        vertices = accomodate_boundary(xx,yy,boundaries);
    end

    area_vec(i) = polyarea(vertices(:,1),vertices(:,2));
    size_vec(i) = 1;
    clear vertices;
end
```

accommodate_boundary.m

```
function new_vertices = ...
    accomodate_boundary(x_vals, y_vals, bounds)
%-------------------------------------------------------------
%accomodate_boundary    David Barnes    8/21/02
%    new_vertices = ...
%        accomodate_boundary(x_vals, y_vals, bounds)
%
%    Finds the vertices of the intersection between the
%    bounded region of interest and a Voronoi polygon
%    which has one or more vertices outside the region.
%
%    Input:  x_vals -- x values of the polygon vertices
%            y_vals -- y values of the polygon vertices
%            bounds -- the minimum and maximum allowable
%                      x & y values for any polygon vertices.
%                      Give as [minx, maxx, miny, maxy]
%
%    Output: new_vertices -- the vertices of the new polygon
%                            that has one (or more) edges
%                            that coincide with a boundary
%-------------------------------------------------------------

% L is the number of vertices
L = length(x_vals);
% so some later equations are solved correctly
bounds2([1,3]) = bounds([1,3])-1;
bounds2([2,4]) = bounds([2,4])+1;

vertices = [x_vals y_vals];

% a list of indices to all vertices outside the boundary
outOfBounds = unique([find(x_vals < bounds(1))'...
    find(x_vals > bounds(2))' find(y_vals < bounds(3))'...
    find(y_vals>bounds(4))']);

% indicate is 1 at the indices of those vertices
% that lie out of bounds
indicate = zeros(L,1);
indicate(outOfBounds) = 1;

% qq is the number of vertices that lie out of bounds
qq = length(outOfBounds);

% to flag the case where an infinite vertex exists
if find(~isfinite(vertices(:,1)))
    fprintf('error: infinite vertex found')
    return ;
end
```

```
%----------- when only one vertex lies in bounds -----------
one_in = 0;
moved = 0;
if qq == L-1
    one_in = 1;

    % if the in-bounds vertex is not first,
    % permute the vertices until it is
    if indicate(1) == 1
        move2 = 1;
        perm2 = [zeros(L-1,1), eye(L-1); 1, zeros(1,L-1)];
        count = 0;

        while move2
            vertices = perm2*vertices;
            indicate = perm2*indicate;

            if indicate(1)==0
                % end loop
                move2 = 0;
                moved = 1;
            end

            count = count + 1;
            if count>100
                fprintf('Error: infinite loop...\n')
                break
            end

        end
    end
    % then augment the vertices by repeating the in-bounds
    % vertex in the last position
    vertices(L+1,:) = vertices(1,:);
    L = L+1;
    indicate(L) = 0;
    count = 0;
end
%-----------------------------------------------------------

%------ when the out of bounds vertices are the first ------
%------------- and/or last vertices in the list ------------
ends = outOfBounds([1,qq]);
if ends(1)==1 | ends(2) == L
    move = 1;
    moved = 1;
    perm = [zeros(L-1,1), eye(L-1);1,zeros(1,L-1)];

    % permute the order of the vertices those that lie out
    % of bounds are not at the beginning or end of the list
    power = 0;
```

```
    count = 0;
    while move
        vertices = perm*vertices;
        indicate = perm*indicate;
        % keep track of how vertices were shifted
        power = power + 1;

        if (indicate(1) ==0) & (indicate(L) == 0)
            % if first and last vertices are in bounds
            % end loop
            move = 0;
        end

        % safety catch
        count = count +1;
        if count>100
            fprintf('Error: infinite loop...\n')
            break
        end
    end
end
%-------------------------------------------------------------

%----- find the intersection of the edges of the polygon ---
%----------- and the boundary imposed on the space ---------

% get indices of out-of-bounds vertices
indexx = find(indicate);
LL = length(indexx);

% get the polygon vertices that define the line that
% crosses the boundary
x1 = [vertices(indexx(1),1) vertices(indexx(LL),1)];
x2 = [vertices(indexx(1)-1,1) vertices(indexx(LL)+1,1)];
y1 = [vertices(indexx(1),2) vertices(indexx(LL),2)];
y2 = [vertices(indexx(1)-1,2) vertices(indexx(LL)+1,2)];

% find the lines defined by those vertices
line{1} = [(y1(1)-y2(1)), (x2(1)-x1(1)), ...
        (((x2(1)-x1(1))*y1(1) + (y1(1)-y2(1))*x1(1)))];
line{2} = [(y1(2)-y2(2)), (x2(2)-x1(2)), ...
        (((x2(2)-x1(2))*y1(2) + (y1(2)-y2(2))*x1(2)))];

% equations of the boundary lines
boundeq{1} = [1 0 bounds(1)];
boundeq{2} = [1 0 bounds(2)];
boundeq{3} = [0 1 bounds(3)];
boundeq{4} = [0 1 bounds(4)];

%----- for the two systems of equations, find solutions ----
for j = 1:2
```

```
    % the line could cross any of the four boundaries
    for i = 1:4
        newm{i} = rref([boundeq{i};line{j}]);
        sol_1(:,i) = newm{i}(:,3);
    end

    % order the x & y coordinates from the vertices
    % of the polygon
    aa = sort([x1(j) x2(j)]);
    bb = sort([y1(j) y2(j)]);

    % take the solution to the equation that lies on the
    % line that crosses the boundary, and that lies within
    % the boundary
    xs = find(aa(1) <= sol_1(1,:) & ...
        sol_1(1,:) <= aa(2) & bounds2(1) <= sol_1(1,:) & ...
        sol_1(1,:) <= bounds2(2)&bb(1) <= sol_1(2,:) & ...
        sol_1(2,:) <= bb(2) & bounds2(3) <= sol_1(2,:) & ...
        sol_1(2,:) <= bounds2(4));

    % if a solution is found, assign it as one of the new
    % vertices of the polygon
    if xs
        new_v{j} = sol_1(:,xs(1))';
    else
        fprintf('Error: no solution to boundary equation\n')
        return
    end
    clear newm sol_1 new_ind xs;
end
%-------------------------------------------------------------

newx = [new_v{1}(1) new_v{2}(1)];
newy = [new_v{1}(2) new_v{2}(2)];

% check to see if a corner is cut off by the two new
vertices
if sum((find(newx == bounds(1) | newx == bounds(2)))) & ...
    sum((find(newy == bounds(3) | newy == bounds(4))))
    % then the cell contains a corner, so find the corner
    x_loc = find(newx == bounds(1) | newx == bounds(2));
    xcorner = newx(x_loc);
    y_loc = find(newy == bounds(3) | newy == bounds(4));
    ycorner = newy(y_loc);
    cornerv = [xcorner ycorner];
    % insert the new vertices and the corner in the
    % non-out-of-bounds vertices
    frst = vertices(1:indexx(1)-1,:);
    lst = vertices(indexx(LL)+1:length(vertices),:);
    vertices = [frst;new_v{1};cornerv;new_v{2};lst];
```

```
else
    % insert the new vertices in the proper place in the
    % n x 2 matrix of polygon vertices
    if LL == 1
        % only one vertex was out of bounds
        frst = vertices(1:indexx(1)-1,:);
        lst = vertices(indexx(LL)+1:length(vertices),:);
        vertices = [frst;new_v{1};new_v{2};lst];
    else
        % replace the first and last out of bounds
        % vertices
        vertices(indexx(1),:) = new_v{1};
        vertices(indexx(LL),:) = new_v{2};
    end

    if one_in
        % if the original vertices were augmented
        vertices(L,:) = [];
    end

    if LL>2
        % delete all out-of bounds vertices that still
        % remain in the matrix
        vertices(indexx(2:LL-1),:) = [];
    end
end

% set return value
new_vertices = vertices;
```

initializematrices.m

```
function [merge_mat, contains, blocknum, temparea,...
        tempsize, tempadj] = initializematrices(pref)
%-------------------------------------------------------------
%initializematrices     David Barnes    9/17/02
%    [merge_mat, contains, blocknum, temparea,...
%        tempsize, tempadj] = initializematrices(pref)
%
%    A function to initialize the arrays for two dimensional
%    bayesian block analysis.  Begins a top-down or
%    bottom-up approach.
%
%    input:  pref - option for different starting states
%                   1 for bottom-up (complete partition)
%                   2 for top-down (single block partition)
%
%    output: merge_mat - a matrix of merge/divide factors
%                        m(i,j) = merge factor i ~= j
%                        m(i,j) = divide factor i == j
%            contains - cell array; the cells contained in
%                       every block
%            blocknum - blocknum(i) = index of the block to
%                       which cell(i) belongs
%            temparea - areas of blocks
%            tempsize - sizes of blocks
%            tempadj  - adjacency matrix that will be updated
%                       as the block structure changes
%-------------------------------------------------------------

global area_vec size_vec blocknum
global adjsp merge_mat
global C

% initialize array
numPoints=length(area_vec);
merge_mat = sparse(zeros(numPoints));

if pref == 1
    temparea = area_vec;
    tempsize = size_vec;
    tempadj = adjsp;
    blocknum = linspace(1,numPoints,numPoints);

    % compute merge factors for any adjacent cells
    for i=1:numPoints
        newMrg = find(tempadj(i,:));
        newL = length(newMrg);
        merge_mat(i,newMrg) = ...
           mrgfctr2(temparea(i)*ones(1,newL), ...
```

```
            tempsize(i)*ones(1,newL), temparea(newMrg),...
            tempsize(newMrg),C);
    end

    % each cell constitutes its own block
    for i=1:numPoints
         contains{i}=i;
    end
end


if pref == 2
    % set the single block to have index 1
    temparea = zeros(1,numPoints);
    temparea(1) = sum(area_vec);

    tempsize = zeros(1,numPoints);
    tempsize(1) = numPoints;
    % no adjacencies
    tempadj = zeros(numPoints);

    contains = cell(size(tempsize));
    contains{1} = linspace(1,numPoints,numPoints);
    blocknum = ones(1,numPoints);

    % set divide factors for all cells in the block
    newareas = temparea(1)*ones(1,numPoints) - area_vec;
    newsizes = (numPoints - 1)*ones(1,numPoints);
    remove = -mrgfctr2(newareas, newsizes, area_vec, ...
        ones(1,numPoints),C);

    for i =1:numPoints
        merge_mat(i,i) = remove(i);
    end
end
```

mergedivideblocks4.m

```
function [newContents, lables] =...
    mergedivideblocks4(opNumber, row, col, pref)
%-------------------------------------------------------------
%mergedivideblocks4     David Barnes 9/17/02
%   [newContents, lables] =...
%       mergedivideblocks4(opNumber, row, col, pref)
%
%   A function to merge/divide blocks.  Allows combining two
%   blocks to create one new block and removing one cell
%   from a block to create two new blocks.  If a cell v is
%   to be removed, the connectivity of its neighbors is
%   checked.  This is done using the Laplacian matrix of
%   just those neighbors in the block without v.  If the
%   neighbors do not create a connected graph then the
%   removal of the cell is terminated since it may
%   correspond to a cut-vertex of the dual graph for the
%   block it belongs to.  On the first call to this
%   function all the arrays are initialized.  This is
%   done here so that they are persistent, not global.
%   If a cell is to be removed from a block and that
%   block is located at the index for the cell, another
%   index for the block is found and the block is moved
%   there.  The sub function exchange(.) is included to
%   help in these switches.
%
% input:    opNumber - used as a flag the first time the
%                      function is called to initialize
%                      some arrays
%           row, col - the block(s) being operated on
%           pref     - top-down (1) or bottom-up (2) start
% output:   newContents - to update the contents of blocks
%           lables - the blocks to update
%-------------------------------------------------------------
global area_vec size_vec blocknum
global adjsp merge_mat
global C

persistent contains
persistent temparea tempsize
persistent tempadj

% if this is the first iteration, initialize all arrays

if opNumber==1
    % get all the initial information needed
    % done here so variables are persistent, not global
```

```
    [merge_mat, contains, blocknum, temparea, tempsize,...
            tempadj] = initializematrices(pref);
    newContents = contains;
    lables = linspace(1,length(contains),length(contains));

    for i = 1:length(area_vec)
        adjcell{i} = tempadj(i,:);
    end
    return
end

least = row;
maximum = col;

%-------------------------------------------------------------
if row~=col  % merge blocks
    [newContents, temparea, tempsize, contains, ...
        tempadj] = merge(least, maximum, temparea, ...
        tempsize, contains, tempadj);
    lables = [least, maximum];

else % divide block: remove the cell from its block
    bl = blocknum(least);

%-- check if all of cell least's neighbors are connected ---
    if length(contains{bl})>2
        neighborBlocks = blocknum(find(adjsp(least,:)));
        numberOut = length(find(neighborBlocks ~= bl));
        if numberOut > 0
            neighbors = find(adjsp(least,:));
            neigborsInBlock = ...
                intersect(neighbors, contains{bl});
            subadj=adjsp(neigborsInBlock, neigborsInBlock);
            laplacian = diag(sum(subadj)) - subadj;
            [rw cl] = size(laplacian);
            test = rw - rank2(laplacian);

%------ if least corresponds to a cut vertex, return ------
            if test > 1
                merge_mat(least,least) = 0;
                newContents = [];
                lables = [];
                return
            end
        end
    end
```

```
% a block is located at least so move block to another cell
    if bl == least
        possibilities = setxor(bl, contains{bl});
        newbl = possibilities(1);
        tempadj(newbl,:) = tempadj(least,:);
        tempadj(:,newbl) = tempadj(newbl,:)';
        blocknum(contains{least}) = newbl;
        contains{newbl}= contains{least};
        temparea(newbl)=temparea(least);
        tempsize(newbl)=tempsize(least);

        bl = newbl;
    end
%-------------------------------------------------------------
    % set return values
    [newContents, temparea, tempsize, contains, ...
    tempadj] = divide(least,bl, temparea, tempsize, ...
    contains, tempadj);
    lables = [least, maximum];
end

%-------------------------------------------------------------
function [newarray1, newarray2] = exchange(array1, array2)
% a subfunction to exchange arrays or values
newarray1 = array2;
newarray2 = array1;
```

merge.m

```
function [newContents, temparea, tempsize, ...
        contains, tempadj] = merge(least, maximum, ...
        temparea, tempsize, contains, tempadj)
%-------------------------------------------------------------
%merge     David Barnes    9/17/02
%    [newContents, temparea, tempsize, contains, ...
%        tempadj] = merge(least, maximum, temparea, ...
%        tempsize, contains, tempadj)
%
%    Merges two blocks located at least and maximum, and
%    returns updates for the arrays affected.  Always
%    combines cells to the index least.  All the
%    properties of the new block will be contained in the
%    respective arrays at the location least.
%
%    Input:       least - location of the first block
%                 maximum - location of the second block
%                 temparea - areas of all the blocks
%                 tempsize - sizes of all the blocks
%                 contains - cell array; members of all blocks
%                 tempadj - adjacency matrix for current
%                           block structure
%
%    Output:      newContents - updates for contents in parent
%                               algorithm
%                 temparea - updated areas of all blocks
%                 tempsize - updated sizes of all blocks
%                 contains - cell array; members of all blocks
%                 tempadj - updated adjacency matrix for
%                           current block structure
%
%-------------------------------------------------------------

global area_vec size_vec blocknum
global adjsp merge_mat
global C

% merge into the cell with the index least
temparea(least) = temparea(least) + temparea(maximum);
tempsize(least) = tempsize(least) + tempsize(maximum);

% set unused cell to zero in temparea and tempsize
temparea(maximum)=0;
tempsize(maximum)=0;

% update what cells are contained in the new block
contains{least}=[contains{least} contains{maximum}];
contains{maximum}=[0];
```

```
% update the block number for each cell
blocknum(contains{least}) = least;

% the new block is adjacent to any block either of
% least or maximum was adjacent to
newA = find(tempadj(maximum,:));
tempadj(least, [newA]) = 1;
tempadj([newA], least) = 1;
tempadj(least,least) = 0;
tempadj(maximum,:) = 0;
tempadj(:,maximum) = 0;

% invalid merge factors deleted
merge_mat([least,maximum],:) = 0;
merge_mat(:,[least,maximum]) = 0;

% update merge factors based on new adjacency info
newMerge = find(tempadj(least,:));
newmergefactors(least, newMerge, temparea(least), ...
    tempsize(least), temparea(newMerge),...
    tempsize(newMerge))

% compute effect of removing each cell from this block
members = contains{least};
divideMatrix = getDivideFactors(members,...
    temparea(least), tempsize(least));

newContents{1} = [contains{least}];
newContents{2} = [0];
```

divide.m

```
function [newContents, temparea, tempsize, contains, ...
    tempadj] = divide(least, pblock, temparea, ...
    tempsize, contains, tempadj)
%-------------------------------------------------------------
%divide     David Barnes    9/17/02
%    [newContents, temparea, tempsize, contains, ...
%        tempadj] = divide(least, pblock, temparea, ...
%        tempsize, contains, tempadj)
%
%    Removes a single cell from a block and returns updates
%    for the arrays affected.  Only allows the removal of
%    a cell from a block that remains connected after the
%    operation.  Removing a cell that corresponds to a cut
%    vertex in the dual graph of the tessellation will
%    result in an invalid division (disjoint blocks will
%    continue to be considered a single block).
%
%    Input:      least - the cell to be removed
%                pblock - location of the parent block
%                temparea - areas of all the blocks
%                tempsize - sizes of all the blocks
%                contains - cell array; members of all blocks
%                tempadj - adjacency matrix for current
%                          block structure
%
%    Output:     newContents - updates for contents in parent
%                              algorithm
%                temparea - updated areas of all blocks
%                tempsize - updated sizes of all blocks
%                contains - cell array; members of all blocks
%                tempadj - updated adjacency matrix for
%                          current block structure
%
%-------------------------------------------------------------

global area_vec size_vec blocknum
global adjsp merge_mat
global C

% reset areas & sizes for cell removed & remaining block
temparea(least) = area_vec(least);
tempsize(least) = 1;
temparea(pblock) = temparea(pblock) - temparea(least);
tempsize(pblock) = tempsize(pblock) - 1;

% least is now its own block/no longer contained in pblock
contains{least} = least;
contains{pblock} = setxor(least, contains{pblock});
```

```
blocknum(least) = least;
blocknum(contains{pblock}) = pblock;

% least now adjacent to any block containing one of its
% original neighbors
newadj = find(adjsp(least,:));
tempadj(least,:) = 0;
tempadj(:,least) = 0;
tempadj(least,blocknum(newadj)) = 1;
tempadj(blocknum(newadj),least) = 1;

% reset merge factors for cell removed
newMerge = find(tempadj(least,:));
newmergefactors(least, newMerge, temparea(least), ...
    tempsize(least), temparea(newMerge),...
    tempsize(newMerge))

% reset merge factors for remaining block
newMerge2 = find(tempadj(pblock,:));
newmergefactors(pblock, newMerge2, temparea(pblock), ...
    tempsize(pblock), temparea(newMerge2),...
    tempsize(newMerge2))

% reset divide factors for all cells in pblock
members = contains{pblock};
if length(contains{pblock})>1
    divideMatrix = getDivideFactors(members,...
        temparea(pblock), tempsize(pblock));
else
    merge_mat(contains{pblock},contains{pblock}) = 0;
end
merge_mat(least,least) = 0;

newContents{1} = [contains{least}];
newContents{2} = [contains{pblock}];
lables = [least, pblock];
```

newmergefactors.m

```
function newmergefactors(index, ...
    newMerge, iarea, isize, areasIn, sizesIn)
%-------------------------------------------------------------
%newmergefactors     David Barnes 9/17/02
%   newmergefactors(index, ...
%       newMerge, iarea, isize, areasIn, sizesIn)
%
%   A function to update merge factors in the merge matrix
%   no output is returned since merge_mat is global
%
%   input:  index - the row & col of merge_mat being updated
%           newMerge - blocks adjacent to block at index
%           iarea - area of block at index
%           isize - size of block at index
%           areasIn - areas of blocks adjacent to index
%           sizesIn - sizes of blocks adjacent to index
%
%   output: none - updates made to global array merge_mat
%
%-------------------------------------------------------------

global merge_mat
global C

% clear previous entries
merge_mat(index,:) = 0;
merge_mat(:,index) = 0;

nFactors = length(areasIn);

% if one or more blocks adjacent to index, update factors
if nFactors >=1
    merge_mat(index,newMerge) = ...
        mrgfctr2(iarea*ones(1,nFactors), ...
        isize*ones(1,nFactors), areasIn, sizesIn, C);
        merge_mat(:,index) = merge_mat(index,:)';
end
```

getDivideFactors.m

```
function divideMatrix = ...
    getDivideFactors(members, cellarea, cellsize)
%-------------------------------------------------------------
%getDivideFactors   David Barnes 9/17/02
%   divideMatrix = ...
%       getDivideFactors(members, cellarea, cellsize)
%
%   A function to compute the divide factors for all members
%   of a block.  A division is simply the removal of a
%   single cell from a parent block.  If a cell corresponds
%   to a cut vertex for a block in the dual graph of the
%   relevant subset of the tessellation it may not be
%   removed as a valid division of this kind.  Here the
%   merge factors for such divisions are still computed.
%
%   input:  members - the cells that belong to a block
%           cellarea - area of the block
%           cellsize - size of the block
%
%   output: divideMatrix - a matrix of all zeros exept for
%                          updates to be made for divide
%                          factors on the diagonal
%
%-------------------------------------------------------------
global merge_mat
global area_vec size_vec
global C

numMembers = length(members);
% the divide factors that need to be updated
oldDivideFactors = diag(merge_mat(members,members));
oldDivideVector = sparse(zeros(length(area_vec),1));
newDivideVector = oldDivideVector;
oldDivideVector(members) = oldDivideFactors;
% find the areas of all potential blocks resulting from
% the removal of a single cell
newareas = cellarea*ones(1,numMembers) - ...
    area_vec(members);
newsizes = (cellsize - 1)*ones(1,numMembers);
% generate the new divide factors
remove = -mrgfctr2(newareas, newsizes, ...
    area_vec(members), ones(1,numMembers), C);

newDivideVector(members) = remove;
newDivideVector = newDivideVector - oldDivideVector;
divideMatrix = sparse(diag(newDivideVector));
% update divide factors in the merge matrix
merge_mat = plus(merge_mat,  divideMatrix);
```

mrgfctr2.m

```
function m = mrgfctr2(volume1,size1,volume2,size2,nprior)
%-------------------------------------------------------------
%mrgfctr2.m    David Barnes 7/02/02
%   m = mrgfctr2(volume1, size1, volume2, size2, nprior)
%
%   Computes the effect merging two blocks has on the log of
%   the objective function
%
%   Input:  volume1 - volume of the first block
%           size1 - size of the first block
%           volume2 - volume of the second block
%           size2 - size of the second block
%           nprior - a prior on the number of blocks
%
%   Output: m - the merge factor for the two blocks
%-------------------------------------------------------------

VAL_IF_MERGE = ...
    glblpost2(volume1 + volume2, size1 +size2, nprior);

P1 = glblpost2(volume1, size1, nprior);
P2 = glblpost2(volume2, size2, nprior);

m = VAL_IF_MERGE - P1 - P2;
```

glblpost2.m

```
function value = glblpost2(V, S, n_prior)
%-------------------------------------------------------------
%glblpost2.m   David Barnes 7/02/02
%   value = glblpost2(V, S, n_prior)
%
%   Objective function with a prior on the number of blocks
%
%   Input:  V - the area of a block
%           S - the size of a block
%           n_prior - a prior on the number of blocks
%
%   Output: value - objective function value of the block
%-------------------------------------------------------------

x = V-S+1;
loc = find(x<0);

if loc V(loc) = 100000;
end

if V<S-1 value = 0;
else
    value = GAMMALN(S+1) + GAMMALN(V-S+1) -...
        GAMMALN(V+2) + n_prior;
end

if loc value(loc) = 0;
end
```

colour2d5.m

```
function [T, newa_vec, news_vec, contains, posterior] = ...
    colour2d5(data, alg, inParam, onOff, const, nBlockPrior)
%-------------------------------------------------------------
%colour2d5     David Barnes    9/18/02
%    [T, newa_vec, news_vec, contains, posterior] = ...
%    colour2d5(data,alg,parameters, onOff, const,nBlockPrior)
%
%    A function that gives graphical output for the 2-d
%    algorithms.  Shades regions of uniform intensity with
%    the same color.
%
%    input:  data -  an nx2 matrix whose rows are the
%                    ordered pairs that determine data points
%            alg  -  allows user to choose from:
%                1 -> greedy merge only
%                2 -> greedy merge/divide
%                3 -> tabu search
%                4 -> simulated annealing m/d
%            inParam:
%                [0] - BM2b_2
%                [pref] - BMD2b_2
%                [pref, tabu, cycles] - tabu2db_2
%                [pref, T, cycles, LK, dec, incr] - SA2db_3
%            onOff - switches on/off parts of function
%                [1 0 0] - wait for keyboard response before
%                          displaying each new block
%                [0 1 0] - show voronoi diagram in final
%                          display
%                [0 0 1] - generate a new figure each
%                          time a new block is added, display
%                          all current blocks in the new fig
%            const - distance (in x and y directions) the
%                    rectangular boundary will be from the
%                    minimum and maximum x and y data values
%            nBlockPrior - a prior on the number of blocks
%                          in the model
%    output: displays a figure with distinct blocks shaded
%            with different colors
%            T -     time for algorithm to find a solution
%            newa_vec - areas of the blocks returned
%            news_vec - sizes of the blocks returned
%            contains - cell array of voronoi cells
%                       contained in each block
%            posterior - the objective function value for
%                        the partition returned
%-------------------------------------------------------------
```

```
L=length(data);
min_x = min(data(:,1)) - const;
max_x = max(data(:,1)) + const;
min_y = min(data(:,2)) - const;
max_y = max(data(:,2)) + const;
boundaries = [min_x max_x min_y max_y];
const2 = 100*const;
% four points imposed on the region outside one containing
% all the data so no unbounded voronoi cells are generated
borders = [min_x min_y; min_x max_y;...
        max_x min_y; max_x max_y] + ...
        const2*[-1 -1;-1 1;1 -1 ;1 1];
global area_vec size_vec
global adjsp
global C
C = nBlockPrior;
[area_vec, size_vec, adjsp] =...
    condition2d(data, boundaries, borders);
%-------------------------------------------------------------
switch alg
    case 1
        [T, newa_vec, news_vec, contains, posterior]=...
            BM2(area_vec, size_vec, adjsp, C);
    case 2
        [T, newa_vec, news_vec, contains, posterior] = ...
            BMD2(area_vec, size_vec, adjsp, inParam(1),C);
    case 3
        [T, newa_vec, news_vec, contains, posterior]=...
            tabu2d(area_vec, size_vec, adjsp, inParam(1),...
            inParam(2), inParam(3),C);
    case 4
        [T, newa_vec, news_vec, contains, posterior] = ...
            SA2d(area_vec, size_vec, adjsp, inParam(1), ...
            inParam(2), inParam(3), inParam(4), ...
            inParam(5), inParam(6),C);
    otherwise
        fprintf('Error: invalid algorithm requested')
        return
end
%-------------------------------------------------------------
[v,c]=voronoin([data;borders]);
bknum=length(newa_vec);
hts = news_vec./newa_vec;
mult = fliplr([linspace(0.1,1,bknum)].^2);
order  = sort(hts);
order2  = sort(newa_vec);
% to white out background
noiseD = find(newa_vec == order2(bknum))
colours{noiseD} = [1 1 1];
hold on
%-------------------------------------------------------------
```

```
for i=1:bknum
    intense = hts(i);
    index = find(order==intense);
    if i ~= noiseD
        colours{i} = [1*(1-mult(index(1))), ...
            1*(mult(index(1))), 0];
    end
    cells = contains{i};        % list cells in block(i)
    if cells
        bksize = length(cells);
        bounds = 0;
        if onOff(1)
            fprintf('Creating a new block of color: ');
            disp(colours{i})
            pause
        end
        if onOff(3)
            figure
            hold on
        end
        for j=1:bksize
            % indices = an index to rows of v (vertices)
            indices=c{cells(j)};

            % use indices to generate a matrix of vertices
            xx = v(indices,1);
            yy = v(indices,2);

            if (min_x <= xx) & (xx <= max_x) & ...
                    (min_y <= yy) & (yy<=max_y)
                % if no vertex is out of the rect. region
                fill(xx, yy, colours{i});
            else
                % vertex is not within the bounded region
                vert = accomodate_boundary(xx,...
                    yy, boundaries);
                fill(vert(:,1), vert(:,2),colours{i});
            end
            axis(boundaries);
            vertices = [];
            clear indices;
        end
    end
    clear cells;
end
%-------------------------------------------------------------
hold on
if onOff(2)
voronoi([data(:,1);borders(:,1)],[data(:,2);borders(:,2)])
end
clear global
```

References

[AK] Emile Aarts, and Jan Korst, *Simulated Annealing and Boltzman Machines*, John Wiley & Sons, New York, 1989.

[Bi] Norman Biggs, *Algebraic Graph Theory*, Cambridge University Press, New York, 1996.

[CL] Gary Chartrand and Linda Lesniak, *Graphs and Digraphs*, Chapman & Hall, New York, 1996.

[HS] T.C. Hu and M.T. Shing, *Combinatorial Algorithms*, Dover Publications, New York, 1982.

[J] Brad Jackson, Sundararajan Arabhi, David Barnes, Alina Alt, Peter Gioumousis, Elyus Gwin, Paungaew Sangtrakulcharoen, Linda Tan, and Tun Tao Tsai, *CAMCOS Project on Cluster Analysis of Astronomical Data: Final Written Report*, SJSU Department of Mathematics and Computer Science, San Jose, 2002.

[Le] Peter M. Lee, *Bayesian Statistics, an Introduction*, Oxford University Press, New York, 1989.

[LM] *Learning Matlab 6*, The Mathworks, Natick, 2001.

[PS] Christos H. Papadimitriou and Kennith Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, 1982.

[Sc98] Jeffrey Scargle, Studies in Astronomical Time Series Analysis V. Bayesian Blocks, A New Method to Analyze Structure in Photon Counting Data, *Astrophysical Journal* 504 (1998), 405-18.

[Sc99] Jeffrey Scargle, *Bayesian Blocks, Divide and Conquer, MCMC, and Cell Coalescence Approaches*, From the 19th International Workshop on Bayesian Inference and Maximum Entropy Methods (MAXENT 1999), Boise, 1999, Eds. J. Rychert, G. Erickson, and R. Smith, AIP Conference Proceedings, 567 (2001), 245-256.

[Sc01] Jeffrey Scargle, *Bayesian Blocks in Two or More Dimensions: Image Segmentation and Cluster Analysis*, Contribution to Workshop on Bayesian Inference and Maximum Entropy Methods in Science and Engineering, (MAXENT 2001), Baltimore, 2001.

[SKM] D. Stoyan, W.S. Kendall, and J. Mecke, *Stochastic Geometry and its Applications*, John Wiley & Sons, New York, 1987.